UiO : Department of Informatics
University of Oslo

Managing WirelessHART

Developing a Network Manager

Kristoffer Jensen and Magnus Evensberget
master thesis spring 2013

Managing WirelessHART

Kristoffer Jensen and Magnus Evensberget

30th April 2013

Preface

This thesis is the result of a 60 point masters project completed at the Programming and Network research group at University of Oslo, Department of Informatics. The work was performed by the students Magnus Evensberget and Kristoffer Jensen during the period 2011-2013. The project was supervised by Professor Stein Gjessing, from the Department of Informatics. We have also been in contact with Niels Aakvaag at SINTEF for valuable information regarding micro controllers as well as Trygve Harvei and Waqas Ikram at ABB concerning the WirelessHART standard. Alan Ott, the developer of HidApi, provided debugging guidance for HidApi issues.
The project is a continuation of a master project ending in 2012, where the students had been developing the foundation of a Network Manager operating in a WirelessHART network. Our work has been to further develop the Network Manager, implementing more advanced mechanisms needed for the manager to operate seamlessly. During the project time we have met many obstacles, but we feel that we have done the best out of it considering the prerequisites.
We would like to thank Stein for his valuable input and guidance during the project. We would also like to thank Niels, Trygve, Waqas and Alan for expert guidance on their respective fields of knowledge. Finally, we would like to thank our co-students Ali Ahmad, Henning Klevjer and Kent Varmedal for encouragement and useful input during the course of the project.

Abstract

Wireless Sensor Networks are a relatively new technology, actively researched by many groups around the world. This project is a continuation of a former project at the University of Oslo, focusing on WirelessHART. Our aim were to gain an understanding of the WirelessHART protocol, before further developing and improving the implementation.

This report provides the reader with an introduction to Wireless Sensor Networks to build a general knowledge. This knowledge is then used to help describe the WirelessHART standard in detail, where functionality and structure of each layer are presented. A WirelessHART network consists of several devices performing different tasks, each introduced in turn. After the WirelessHART protocol we present the hardware and software used to build a test network for evaluating our implementation. The aim of the project changed as the devices used were found unsuitable and out-of-date. We changed the direction of the report to focus on the Network Manager and less on the nodes as they were already thoroughly researched.

The implementation part presents a total restructuring of the Gateway, where communication between the different layers are achieved through interfaces. The Gateway correctly propagates advertise packets, initiating a complete join sequence between a node and the Network Manager. The Network Manager generates routes using a graph routing scheme. These routes are used by devices to transmit packets through the network. Security mechanisms and packet construction are implemented according to the standard, providing a packet structure according to the layered approach. We conclude the report with an evaluation of the project, comparing the results to our project goals.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction

Wireless sensor networks are used in every aspect of our lives today and are therefore required to provide a reliable and secure service, often without any interaction or help from humans over longer time. The thesis explores the WirelessHART standard used in sensor networks where monitoring of industrial processes are required. The thesis is a continuation of the thesis "WirelessHART - Gjennomgang og implementering" by Håvard Tegelsrud and Jørgen Frøysadal, and "Design and Implementations of a Rudimentary WirelessHART Network" by Anders Asperheim, Rune V. Sjøen and Kaja F. L. Skaar.

WirelessHART is implemented on small nodes with limited resources when it comes to storage, processing power and energy. Sensor nodes are small devices capable of gathering and processing sensor measurements and communicate with other devices[1]. As sensor nodes are energy constrained due to their size and small power supply, we need to find a routing protocol that is optimised for these conditions. In the case a device should fail, the network is also required to dynamically repair itself as expected of a robust industrial network.

1.1 Previous Work

Our work is a continuation of the work performed by previous master students at University of Oslo. During the time period of August 2009 and until May 2010 the students Håvard Tegelsrud and Jørgen Frøysadal worked with Statoil in order to develop the foundation of a WirelessHART network running on nodes produced by Atmel. Statoil wanted to use the nodes to communicate between their industrial installations in the North Sea. During this time they were able to develop the code needed on the Physical Layer for the network components to create a working network. During the autumn of 2010, the master students Anders Asperheim, Rune V. Sjøen and Kaja F. L. Skaar continued developing code on the Link Layer. They worked until August 2012, when they submitted their research. At

[1] A device generally consists of a microcontroller, transceiver, memory unit, power source and sensors.

the end they were focusing on the Network Layer and implementing the procedure of packet transmission on the Network Manager.

1.2 Motivation

The overall goal of the project is to implement the fundamental structure required for a functioning WirelessHART network. The project is a continuation of a previous master thesis, so their implementation will be evaluated and function as a platform for further development. The WirelessHART protocol provides a complex network, where portions of the network management is far from optimised. We hope to develop a fully running network, where progress can be made on the general mechanisms such as join process, routing capabilities and security.

1.3 Thesis goal

The goal of our project was to develop the general structures needed to run a WirelessHART network. Parts of the system have been developed by Asperheim et al. in the previous project, so we aim to further implement critical functionality mainly at the Gateway and Network Manager. Our project goals are further specified in sections 5.2.2 and 5.2.3 on pages 65 and 67.

1.4 Thesis relevance

After four years of development on network devices, the thesis of Tegelsrud and Frøysadal has become less relevant. The work done in the previous projects has not made any major progress on the devices, which as a direct consequence has lead to a stalemate while the industry has completed developing the device part. Because of this, our project goals have changed to focus on a field that is largely undiscovered and therefore more interesting. The work performed by major industrial companies on Network Managers has grown over the last years but is far from complete, meaning routing and quality of service (QoS) scheduling is highly relevant for the development area. Progress made during the project can supply this field of development with a useful implementation and information. This makes our master thesis more relevant in the development area if any major progress can be made.

1.5 Interested parties

The following sections provide a brief introduction of the interested parties in the project.

1.5.1 Statoil

Statoil is an international energy-company dealing in oil and gas that were established in 1972 following the discovery of oil in the North Sea. They have operations in 37 countries and their headquarters are in Stavanger, Norway.

Statoil has had a large interest in pushing for the HART and WHART standards, as can clearly be seen by winning the "Hart Plant of the Year 2007" [16]. In order to be nominated for this award, a company needs to present an objective and a solution to complete it. The solution being most cost-efficient and fully utilising the HART protocol is declared the winner. Statoil's objective were to find a way that 24 wells in an oil field could send data to a 1200 kilometre long underseas pipeline, while at the same time minimising the amount of staff needed.
Their solution were to use a distributed control system with secondary controllers closer to the field devices to handle their interfaces. For the full system, 1400 field devices were used for monitoring. A video conferencing application were also used in order for experts to cooperate in discovering and fixing problems bound to arise. As a direct consequence, the HART implementation helped the company meet their challenge for higher efficiency, better availability and safety of their assets through analysing the data. By using the full-time connection provided by the network, Statoil were able to track traffic in order to troubleshoot and diagnose problems instantly instead of waiting for the problem to be discovered at a later point.

1.5.2 Atmel

Atmel is a worldwide corporation that specialises in designing and manufacturing of "microcontrollers, capacitive touch solutions, advanced logic, mixed-signal, non-volatile memory and radio frequency (RF) components" [4]. The company is able to deliver complete solutions to any need within these areas, making them highly desirable for our project. Their headquarters are in San Jose, California, but we had contact with their office in Trondheim, Norway. In section 7.1 we introduce the hardware they provided and chapter 5 elaborates why certain hardware had to be dropped.

1.5.3 ABB

ABB is a multinational company operating in robotics and specifically with power and automation technologies [18] . The company specialises in cost-effective, reliable and environment-friendly solutions [2]. ABB was one of the founding members of HART Communications Foundation, and participated actively with the development of the WirelessHART protocol released in 2004. ABB has developed a fully functioning WirelessHART network, where the sensor devices are proprietary and cannot be modified, while the Network Manager and analyst can be configured according to the needs of the user. The process of including ABB in our project is discussed in chapter 5, and the hardware provided is introduced in section 7.1.1.

1.6 Methodology

Within software development there are several frameworks used to structure, plan and control the development process of an information system. The frameworks are suited for different types of projects, depending on type, scale and development process. The most common frameworks are presented in the following sections.

Waterfall

The Waterfall framework is a sequential development model, where the process is split into clearly separated phases. Once a phase is passed, the phase is over and can not be returned to except for minor instances. The phases are split into *requirements analysis, design, implementation, testing, integration and maintenance.* Due to strict control and separation, the model is unqualified for larger projects.

Prototyping

Prototyping is not a standalone development process, but an addition to handle parts of a larger system developed using incremental, spiral or rapid application development. Prototyping reduces risk by breaking a project into smaller segments to make the system development easier and more flexible. Prototypes are often discarded as they prove to be inefficient, but some turn out to be implemented.

Incremental

The incremental approach breaks a project into smaller parts, where every part goes through a mini-waterfall process before proceeding to the next part. Overall requirements are defined before developing the minor parts.

Spiral

The spiral model combines design and prototyping elements, by separating the process into the phases analysis, evaluation, planning and development. The cycle starts off with determining the objectives, alternatives and constraints of an iteration. Next, every alternative and risk are evaluated. The third part concerns developing and verifying the deliverable. Part four is to plan the next iteration. After one cycle, the phases start over. The framework aims to progress at a steady pace, while maintaining control and reducing risk.

Rapid Application Development

RAD is a combination of iterative development and prototyping. The framework aims for fast delivery and low risk by breaking the project into smaller segments. RAD introduces project control by defining deadlines.

If a deadline is about to slip, the deliverable is reduced rather than extending the deadline. It produces software iteratively, where prototypes are not thrown away. Documentation is attached where necessary for future development.

In our project we followed a pattern similar to the RAD framework. We started out with a set of goals, where every goal were split into smaller development processes. Once a module of the program were implemented, we documented the module and moved on to the next one. We were set back at several occasions described in chapter 5, resulting in reduced deliverables as the deadline could not be extended.

1.7 Software Patterns

Software Patterns are predefined solutions to commonly occurring problems within software development. In this section we will describe the different patterns we decided on implementing, and also mention where the patterns have their uses.

Factory Method The Factory Method pattern is used to implement the factory that deals with the creation of objects without specifying the exact class. The factory can then choose the best fitting object. This pattern is implemented in our queue, sessionFactory and commandFactory systems.

Singleton The Singleton pattern restricts a class from being constructed more than once. This is very useful if you do not want to send objects around in your implementation. We have been using this pattern a lot, in classes like the main layers, queueHandlers and other classes that is initiated once. The basic skeleton of a singleton can be seen in listing 1.1.

Builder This pattern is used to delay the creation of objects, calculation of values or some other process that either is expensive to calculate or is impossible at the time of initialization. We used this pattern for creating objects we do not know what will contain upon creation, like AdvertisementDLPDUs and the command ReportNeighborSignalLevels.

```
1 public class Singleton {
2     private static Singleton singleton = null ;
3     public static Singleton getSingleton() {
4         if(singleton == null)
5         singleton = new Singleton() ;
6         return singleton ;
7     }
8
9     private Singleton() { }
```

```
10 }
```

Listing 1.1: The Singleton pattern

1.8 Thesis overview

The thesis is organised as follows: part two first introduces the user to wireless network technology and terms in order to understand how a network is designed and performs communication between different devices. Next, the most common protocols and standards related to sensor networking are explained. Differences and use-areas are mentioned, before a thorough investigation of the WirelessHART protocol follows. Part three starting at page 25 presents the reader to the development environment, thesis goals, previous work on the topic and lastly what has been implemented throughout the project time. The last part holds an overview of results achieved, along with an evaluation of the results and the project work. A conclusion follows, before future work is suggested at the end.

Part II

Theory

Chapter 2

Wireless networks

The following chapter introduces the reader to fundamental principles in wireless networking. The reader is first presented with the methods Time Division Multiple Access and Carrier Sense Multiple Access for accessing a shared medium. In section 2.1.4 the different topologies of a network are discussed, before general routing protocols are presented in section 2.1.5. To prevent discussing terms several times, some mechanisms are first introduced in chapter 4 WirelessHART on page 23.

2.1 Channel Access

One of the major challenges of wireless networking is detecting and handling collisions when communicating. As nodes in the network are either set to listen for traffic or to send traffic, a node cannot send and receive at the same time. Combined with the fact that all nodes use the same frequency band, only one node in an area can transmit at a time without the risk of creating problems. If more than one node transmit inside the same area[1], a collision might happen and the transmission is corrupted. CSMA (Carrier sense multiple access) and TDMA (Time Division multiple access) are two methods for allowing multiple nodes in a network to transmit data without these collisions happening.

2.1.1 Time Division Multiple Access

Time division multiple access (TDMA) is a method for multiple nodes to use a shared medium based on timeslots. As can be seen on Figure 2.1 the transmission window is divided into timeslots that each node can use. If a node does not need its timeslot, the shared medium will be idle during that nodes timeslot. It is a well-developed technology that does not need contention-periods like CSMA. This gives TDMA a potentially higher utilisation and throughput. TDMA is a straightforward protocol to implement and is suitable for ensuring Quality of Service (QoS). One big problem with this channel access method is that all the node clocks need to be synchronised. This means that overhead will be needed in order to keep

[1] the receiving node are in range of both senders

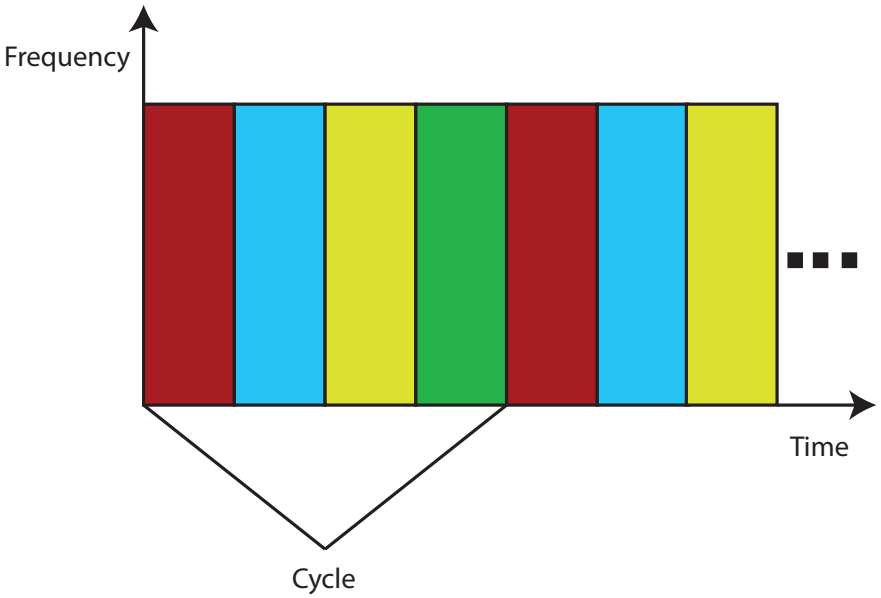

Figure 2.1: A TDMA based network

the clocks up to date. Another problem is to dynamically allocate timeslots to different nodes, particularly when the network grows and the travel time from the Gateway (see section 4.2.3 on page 26) increases. Mobile nodes further complicate the allocation problem.

2.1.2 Carrier Sense Multiple Access

Carrier Sense Multiple Access (CSMA) is a probabilistic medium access protocol that uses different techniques in order to verify the absence of traffic on the shared medium before transmitting. CD (Collision Detection) is used to improve the performance of CSMA by terminating the transmission if a collision is detected, and it also reduces the probability for a collision on the second try. CA (Collision Avoidance) is waiting a random amount of time before retrying transmission if the medium is busy. As radios are not able to listen for traffic while transmitting, Collision Avoidance is the only possibility for wireless communication.

There are several access modes;

- 1-persistent - When the sender node is ready to send, it checks if the medium is busy. If it is busy the node listens continually until the medium is idle before sending its data. If a collision occurs, the sender waits a random amount of time before trying to resend. This is the most widely used version of CSMA/CD, and is used in wired Ethernet like 10BASE5 and early versions of twisted-pair Ethernet.

- P-persistent - This version of CSMA listens continually as 1-persistent, but when the medium becomes idle it is a probability p that the node will try to send. If the node chooses not to send, it waits

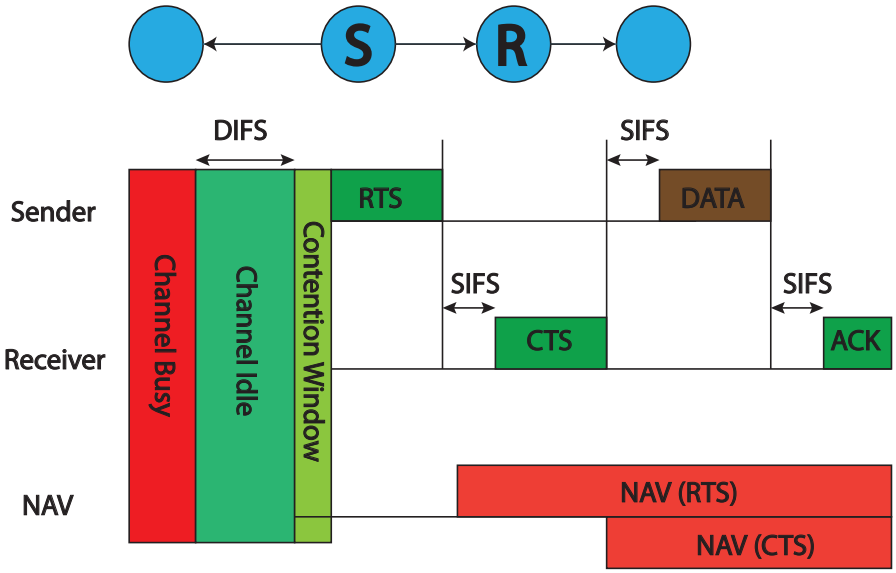

Figure 2.2: A CSMA based network

for the next timeslot. The process is repeated until the node, or another node starts sending on the medium. This is the most widely used version of CSMA/CA, and is used in Wi-Fi and Packet Radio Systems.

- Non-persistent - Like 1-persistent, except that if the medium is busy, the node waits a random amount of time before retrying to send after a collision. This leads to better channel utilisation, but also longer delay than 1-persistent. It will also help save some battery power if it is implemented on a sensor network.

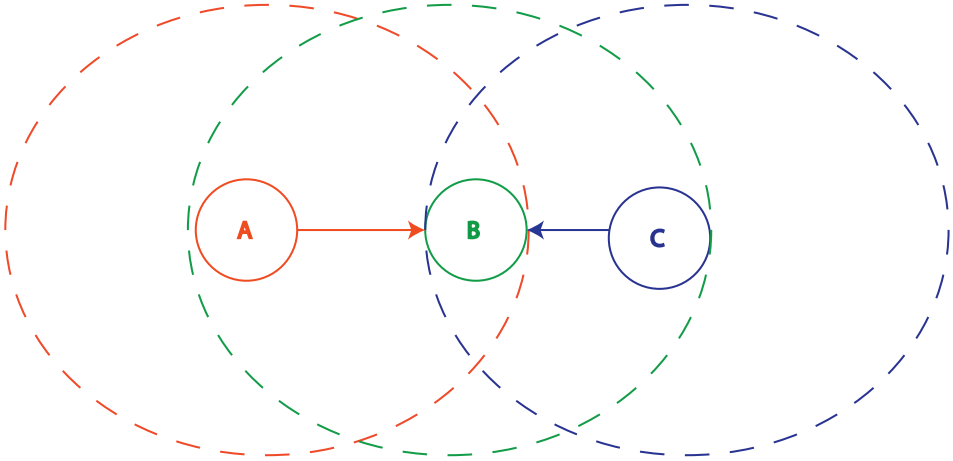

Figure 2.3: Hidden node

There are two problems that must be highlighted when discussing CSMA, namely the "Hidden Terminal Problem" shown in figure 2.3 and the "Exposed Terminal Problem" in figure 2.4.

The Hidden Terminal Problem is present in most wireless networks. If node A and node C want to communicate with node B but are unaware of the each other's existence, they cannot use carrier sense in order to determine if they can send or not as seen in figure 2.3. The solution for this problem is to have two control frames called Ready to Send (RTS) and Clear to Send (CTS) as seen in figure 2.2. The sending node tells all neighbouring nodes that it is about to send a frame with a RTS signal. The receiving node sends back a CTS signal, and all other requests for utilising the link will be denied until the node currently sending is done. This method is called Virtual Carrier Sense. DCF Inter-frame Space (DIFS) are the minimum required waiting time between sensing a busy channel and contention for sending. While Short Inter-frame Space (SIFS) is the time a device has to wait after receiving a packet before responding. Network Allocation Vector (NAV) is the minimum time a node will stay silent after receiving a packet. The timespan depends on the packet detected, as a variable amount of subsequent packets will follow due to the progress in the transmission pattern.

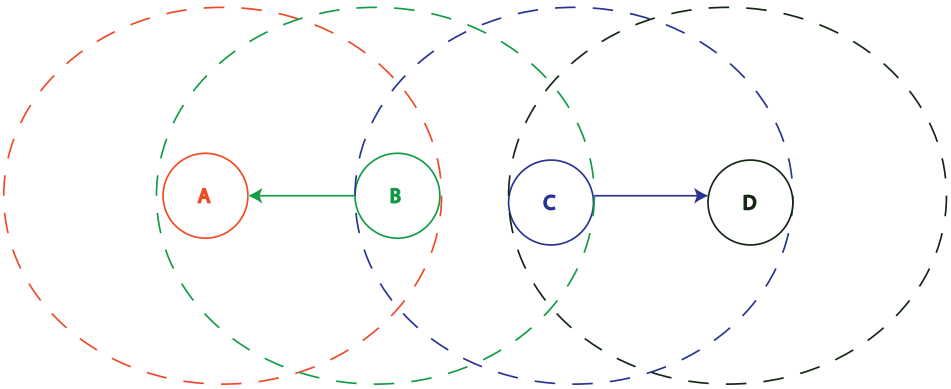

Figure 2.4: Exposed node

The Exposed Terminal Problem (Figure 2.4) makes the node wait unnecessary. Lets say node B starts sending to node A. Node C will see the medium as busy and waits before sending to node D, even though node D is not in range of node B and could receive from C. There is no good solution to this problem. Both Hidden- and Exposed Terminal are lowering overall throughput of the network drastically.

2.1.3 Frequency hopping

Frequency hopping is a technique used by many wireless networks for better channel utilisation. The method rapidly switches between frequency channels in a pseudo-random manner known to the transmitter and the receiver only, distributing the sent packets on different channels. It makes the network much more robust against "narrowband interference" like electromagnetic radiation from a microwave. It also protects against deliberate jamming of the network and is for this reason used in most

military wireless networks like the Norwegian MRR (Multi Role Radio)[2]. For this to work we need good synchronisation of the clocks, which is also mandatory for implementing a TDMA based network.

2.1.4 Network Topologies

When choosing a routing protocol we have to look at the design of the network. Is it a static or mobile topology? How many hops are needed to get from one node to another? There are a lot of topologies we could talk about, but we will mention the most common, namely Ring, Tree, Star- and Mesh Topology.

Star Topology

The Star Network topology is one of the most common network topologies. In its simplest form it consists of a series of nodes with one central node as seen in figure 2.5. Regular Ethernet networks use this topology, where every node in the network is connected to a central node acting as a hub or switch. The central node can either broadcast the packets it receives to all nodes (including the one it got the packet from), or send the packet only on the correct link. These methods are called Passive and Active central node.

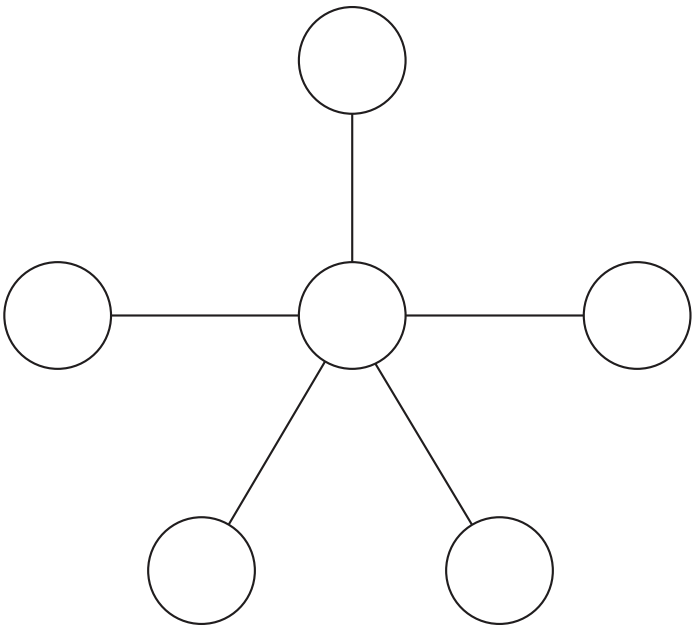

Figure 2.5: Star Topology

The Star network topology has very good performance as it is always two jumps from sender to receiver, but it also puts a lot of overhead on the central node that can create a bottleneck. Another big advantage of the Star topology is that if a node other than the central node disconnects, it

[2]http://www.thalesgroup.com/Press_Releases/naval_PR_Norwegian_Navy_chooses_MRR-3D-NG_radars/

does not affect the network. However, one major disadvantage is that the network is very vulnerable to central point of failure. If this happens, the whole network will break down.

Mesh Topology

The Mesh Network Topology is a more advanced topology that makes nodes in the network not only send information, but also relay information from other nodes. This does however require smarter nodes, and a disconnecting node may reduce performance in the network for some time. The performance is reduced until the network topology is restructured and new routes have been calculated.

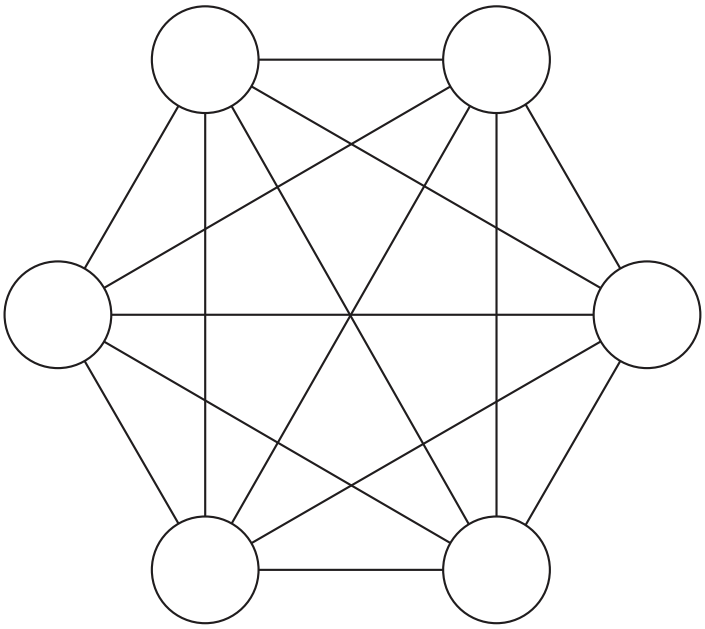

Figure 2.6: Fully connected Mesh Topology

In a Mesh Network you do not need a central controlling node to make the network run. If a node disconnects from the network it is the job of the neighbours to find out that the node has left the network, and relay the information to other nodes in the network.

The Mesh network can take many forms. A fully connected mesh network seen in figure 2.6 is a network where all nodes are connected to all other nodes. This makes the maximal jump distance between two nodes one hop, providing full redundancy and fault tolerance. The disadvantages are that it is expensive and difficult to implement and maintain. A partially connected mesh network seen in figure 2.7, is a network where nodes have to route through other nodes in order to reach certain nodes in the network. At least one node has multiple connections, providing the network with alternative routes in the event of a failure[3]. Mesh topology is used within wired networks as well as wireless.

[3]Unless the fault occurs at the node with multiple connections

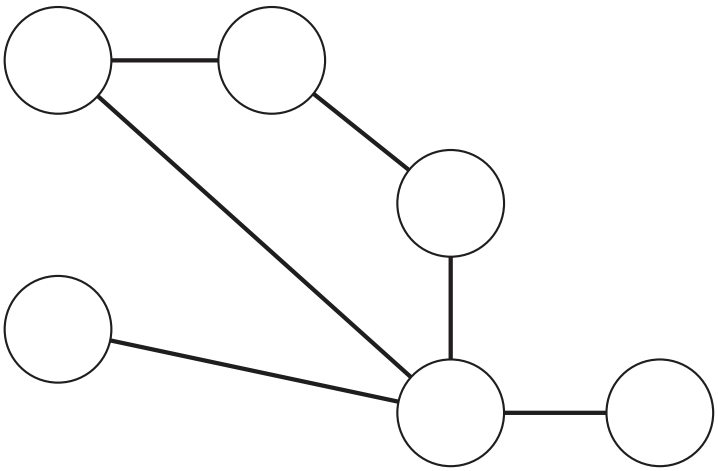

Figure 2.7: Partially connected Mesh Topology

Ring topology

Ring topology is a network topology where every device is connected to two neighbours, forming a chain of devices shaped as a circle. As the network only provides one single pathway to any given node, it is vulnerable to device or link failure as the consequence will be a separated network. Every packet sent will have to go by every node along the way, be processed and sent forward, which means that the topology is slow when maintaining large networks. Compared to star topology, the ring network does not require a server, it performs better under heavy work load and it is cheaper to maintain. Ring topology can be further split into two versions:

Token Ring A token circulates the network. If a node has a packet to send, it seizes the token in order to send. Only the node holding the token are allowed to send.

FDDI Ring Fiber Distributed Data Interface (FDDI) ring is similar to the token ring. FDDI contains two rings opposed to token ring's single ring. One ring goes each way, making the topology fault resistant against a node breaking down.

Tree topology

A tree topology is a network where one node acts as the root, which has children nodes. These nodes have children and so it continues. The tree network requires a minimum of three levels, as two levels would be equivalent to a star network. The nodes at the higher levels are required to perform more work than the nodes at the lower levels due to the connections established. The advantages are that the network is scalable; a new node can connect to an existing node by creating a new link. Every device is connected by point-to-point connections, avoiding any loops. By having a hierarchical approach, the network is easy to manage and correcting errors are fast due to the structure. The disadvantages are that

maintenance can be difficult when the network spans over large areas. The network is also vulnerable to errors at central nodes, making every network partition isolated.

2.1.5 Routing Protocols

In this section we will discuss two routing protocols required by the WirelessHART standard called Graph- and Source routing. These are not the only routing protocols available to Wireless Sensor Networks, but they are important for this thesis. We will define the protocols here and in section 4.6.1 on page 42 we will discuss how these routing protocols are used in WirelessHART.

Graph Routing

Graph routing is a specific term used within WHART on a table-driven routing protocol. This means that a forwarding table of the entire network is located on either some or all the nodes in the network. If the lookup table is located on some, and not all nodes, these few nodes will have the responsibility to tell the other nodes where to send packets. This is the case in WirelessHART where the Network Manager is the central node with information about the network topology.

Source Routing

Source routing gives a single directed route through the network, specified in the packet itself. If the network topology changes during transmission of this packet it will result in a packet loss. Source routing allows the sender to partially or completely determine the route a packet travels through the network. Although the packet is lost if the route breaks down, the routing method provides several advantages. Troubleshooting is made easier, as a timer for every packet sent is kept. The source node can manage network performance by specifically determine routes, preventing congestion on heavily used links. Source routing is a well-known routing protocol widely used in older systems, while Graph routing is less known and restricted to WHART.

2.2 Chapter Summary

In this chapter we have discussed basic mechanisms required in wireless networks. Methods for scheduling and accessing a medium successfully has been presented. Section 2.1.4 provided an overview of the different network topolgies, and what makes one topology favourable over another. Section 2.1.5 detailed how packets were routed through a network in the different protocols available.

Chapter 3

Standards

This chapter presents the most important protocols and standards used within wireless sensor networks. The protocols IEEE 802.15.4, ISA100.11a, Zigbee and WirelessHART are presented in turn. Towards the end of the chapter a comparison between ISA100.11a and WirelessHART is made, before section 3.5 concludes why WirelessHART is most suitable for an industrial sensor network.

3.1 IEEE 802.15.4

IEEE[1] 802 is a collection of IEEE standards for determining local and metropolitan area networks. The standard is concerned with networks where packets of variable size are transmitted, and the services are limited to the Physical Layer and the Data Link Layer. IEEE 802.15 is an extension of 802, working on Wireless Personal Area Networks (WPAN). The standard is further split into seven subgroups, grouped by use and functionality. The IEEE 802.15.4 is the standard for low-rate personal area networks, and provides the foundation for wireless protocols like ZigBee, ISA100.11a and WirelessHART which we will look into later in this chapter. The IEEE 802.15.4 only defines the Physical Layer and the media access control, so it is up to the different protocols to define the upper layers on the stack as can be seen in figure 3.1.

The standard offers low-cost and low speed communication between nearby devices, which requires no underlying infrastructure to operate. Standard transfer rate is 250 Kbit/s but tradeoffs for gaining lower power requirements can be obtained by utilising one of several different Physical Layer options. This can lead to speeds at either 20 or 40 Kbit/s, which in turn will offer even lower energy consumption. The standard aims to offer devices with low production and operation costs, which are technologically simple with few supported operations. At the same time it is important not to sacrifice flexibility that will allow its use in different settings.

Despite being simple, IEEE 802.14.5 offers several technical functions like reservation of timeslots, Carrier Sense Multiple Access Collision

[1] Institute of Electrical and Electronics Engineers

Avoidance (see section 2.2), security for safe communications and possibilities for managing power through link quality and energy detection. These technical terms were discussed in section 2.1.

3.2 ISA100.11a

ISA100.11a is a wireless network technology developed by the members of the International Society of Automation in 2008. The development was an open process, which meant that everyone could contribute; ranging from the companies that were already members to users and suppliers. Everyone could participate, enabling experts' involvement from every aspect. The work yielded a successful protocol design in the same manner as IEEE 802.15.4. The mission stated by the committee were to "decrease the time, costs, and risks of developing and deploying standards-based industrial wireless devices and systems"[11]. The standard is supposed to provide interoperability, compliance, tools, technical support, education and market awareness for its users, which it largely provides by being dynamic and flexible.

3.3 Zigbee

Zigbee is a wireless protocol developed and ratified by members of the Zigbee alliance, consisting of over 300 interested parties. The radio is based on the 802.15.4 specification and operates on the 2.4 GHz, 900 MHz and 868 MHz bands. Zigbee is often compared with Bluetooth, but is considered a lighter version as it requires less software [20]. Zigbee does not provide functionality such as time slots or security however, which makes it more suitable for home-automation, surveillance and control than industrial use. To improve usability, Zigbee pro was introduced in 2007 and provides channel agility for better utilisation of the communication medium.

3.4 Merging of ISA and WirelessHART

ISA100.11a and WirelessHART are often compared as they provide many of the same functionalities. ISA100.11a consists of routing and non-routing devices, handheld, backbone devices which can operate as a router, gateways, system manager and a security manager, just like WHART. The differences are more apparent on the logical layer, as WirelessHART only supports the HART standard while ISA100.11a supports several; WirelessHART, ZigBee, RFID and more. ISA can handle many application protocols like FOUNDATION fieldbus[2] and HART due to a universal Gateway.

There has been attempts of combining WirelessHART and ISA 100.11a in a global IEC standard [10]. The standard applies to the same application

[2]Communication system serving as the ground-level network in a factory automation environment

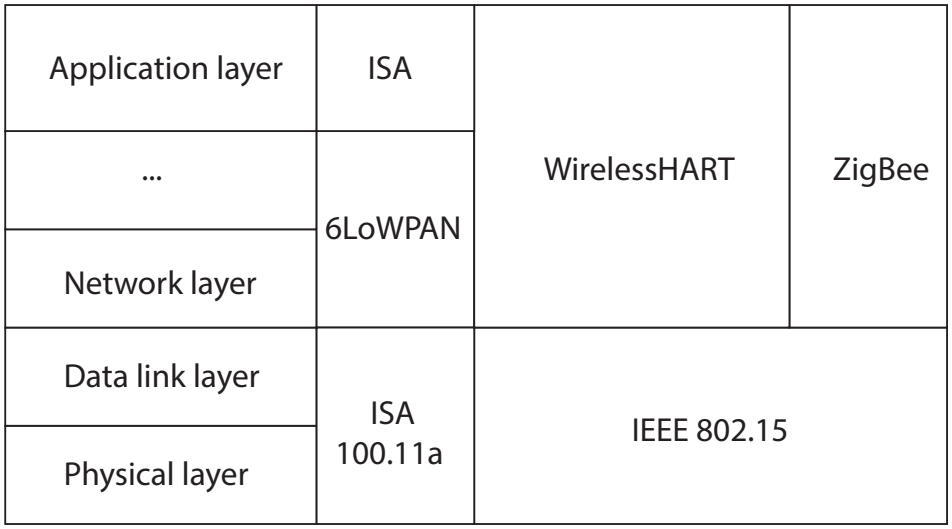

Figure 3.1: The different protocols

areas and uses the same communication protocols based on IEEE 802.15.2. The merging proves to be hard however. Both standards are supported by major companies, making the process of choosing technology hard and tedious as both sides favour their technology.

3.5 Why WirelessHART

It can be discussed which protocol is most favourable, however a few conclusions can be made. While the protocols use the same communication standards, Zigbee is more suitable for non-industrial sensor networks [14]. Because the protocol does not support frequency hopping, it is ineligible due to lack of robustness. Security is also at minimum, which is an essential factor for larger industrial networks in order to protect the integrity and confidentiality of installations. The protocol only supports star topology with 7 slaves per master which is not satisfactory for industrial process automation, compared to WHART's star and mesh topology. As seen in table 3.1 Zigbee does not provide the same quality of service as WirelessHART.

	Zigbee	WirelessHART
Robustness	Low	High
Co-existence	Low	High
Power Consumption	High	Low
Security	Low	High

Table 3.1: Protocol service comparison

ISA100 is built for industrial networks and supports several different protocols. Native HART is not supported however, and the standard

provides its own Physical Layer whereas WirelessHART is built upon the IEEE 802.15 protocol, as seen in figure 3.1. Support for existing HART networks is a critical factor, and alone makes WirelessHART favourable. Compared to Zigbee, the standard is dominant within industrial use as seen in table 3.1.

3.6 Chapter Summary

In this chapter the reader has been introduced to the wireless standards IEEE 802.15.4, ISA100.11a, Zigbee and WirelessHART. The different specifications of each protocol have been highlighted, and compared against the opposing protocols. A section has been dedicated to the ongoing work of merging ISA and WirelessHART, before section 3.5 concludes why WirelessHART is the first choice for running a wireless sensor network.

Chapter 4

WirelessHART

The following chapter presents the reader with the fundamentals of the WirelessHART protocol. Section 4.1 introduces general information around the protocol, followed by an introduction to the different devices running in a WirelessHART network. Section 4.3 discusses how communication between the layers are achieved, while the functionality of the different layers are presented from section 4.4 to section 4.7 on pages 31 - 47. In sections 4.8 and 4.20 the process of time synchronising a network and joining a new device are elaborated.

4.1 The WirelessHART protocol

The following sections introduce the reader to the WirelessHART standard and the old Wired HART protocol.

4.1.1 HART Communication Foundation

The HART Communication Foundation (HCF) is a non-profit international organization that was founded by members of the HART User Group in March 1993, in order to support and promote the use of the HART protocol [8]. The aim of the organization was, and still is "to advance the application of HART technology, strengthen its position in the global market, and help users maximize the value of their smart instrumentation investments". The HCF can be traced back to September 1990 when representatives from 26 companies met and established the HART User Group. HCF has worked with improving the Wired HART standard since the start, but introduced the WirelessHART standard as late as 2007.

4.1.2 The WirelessHART standard

The WirelessHART technology is based on the older HART Communication Protocol standard released in 1993 [8]. Companies saw the need for a wireless standard specifically designed to meet the needs industrial environments provided. Supporting the development of a wireless standard also became important as cabling over long distances proved to be very

costly. The protocol provides key capabilities opposed to standard wireless protocols. WHART provides reliability when faced with interference from outside sources[1], or when nodes and links break down. The mechanisms providing these services such as channel hopping, mesh networking and time synchronized messages are described in sections 4.5 and 4.6. The protocol also provides low power consumption by having few commands and effectively using time slots.

4.1.3 WirelessHART compared to HART

As WirelessHART is the successor of HART, they have certain features in common while differing on other aspects. On the Application Layer they both operate on command-basis with predefined data types discussed in section 4.7. As WHART is a wireless routing protocol, power is an issue and the Network Layer provides power-optimisation and redundant paths for mesh networking while HART only provides physical links. WHART utilises time synchronisation and frequency hopping for sharing the communication medium, while HART uses a token passing function where the nodes follow a master/slave protocol[2]. WHART uses the IEEE 802.15.4 2.4 GHz broadband for transmitting data, where HART uses analogue and digital signalling with a 4-20 mA wire.

4.2 Network components

A WHART network is comprised of five different types of devices, as can be seen in figure 4.1 on the facing page. We will look into all of them in turn in the following sections. However they all share some characteristics. All devices connected to the WirelessHART network are Field Devices, Adapters, Routers, Access Points or Handheld Devices. They are all able to act as routers, sending and receiving packets for other nodes.

4.2.1 Field Devices

The field devices are the basic components in a WHART network. They monitor and store certain data which they send onward when a timeslot permits. The field devices are often placed in spots that are difficult to access for humans, resulting in no human interaction for long periods of time. For this reason it is important that the devices are energy efficient. The devices have a simple, minimalistic interface with few commands. They need to have basic routing capabilities, such as sending and receiving packets between neighbours. This information is provided by the packet headers, which is provided by the Network Manager when the packet is created and sent.

[1] Data traffic sent on same channel from non WHART devices

[2] One device sends and all other listens

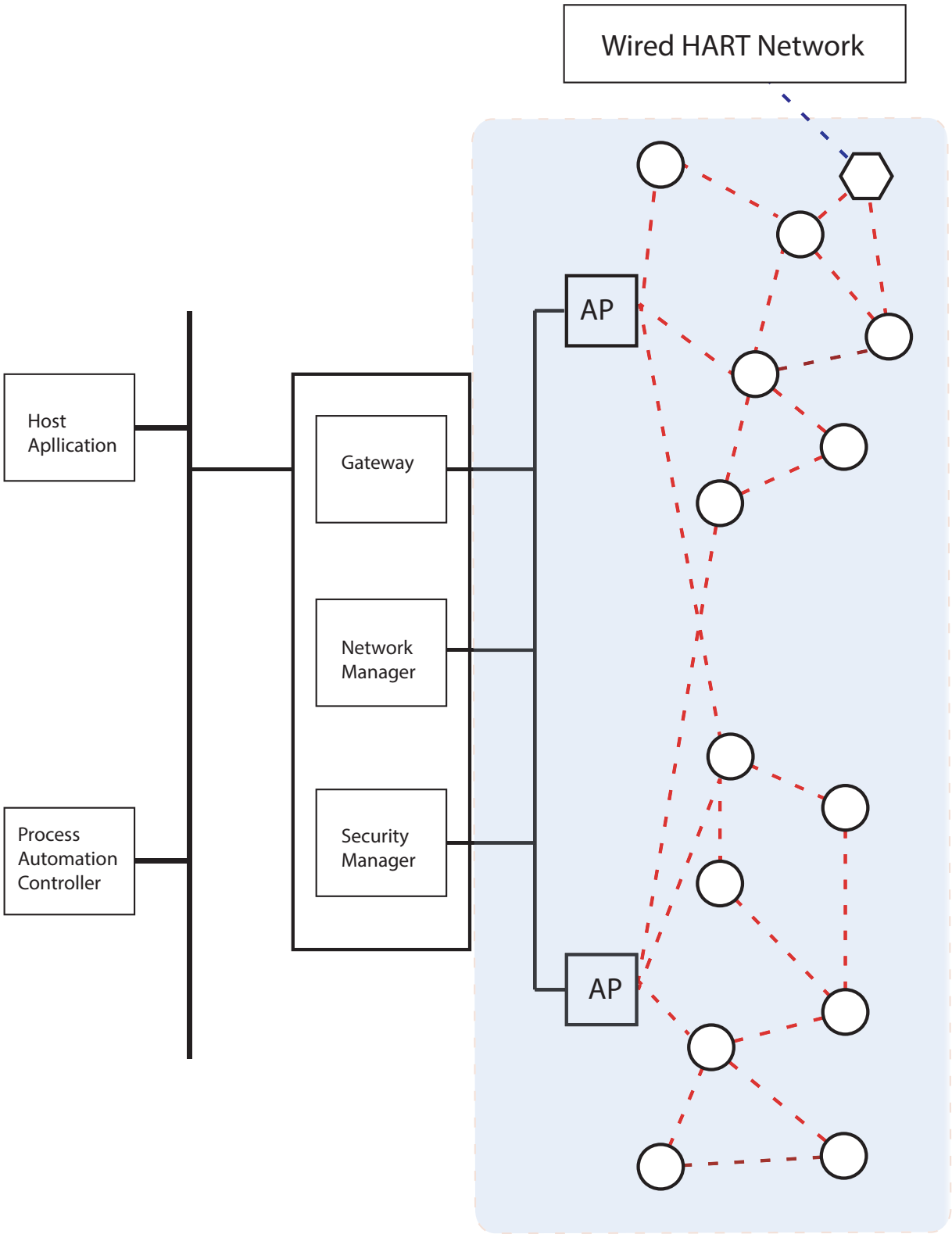

Figure 4.1: a WHART network

4.2.2 Adapter

The adapter is a device that integrates a wired HART network into a WHART network. As wired HART is close to 20 years older than WHART, many wired HART networks already exist. It is desirable to utilise WHART when expanding these networks, while still keeping the old infrastructure. The adapters make certain that the process of converting data from wired to wireless transmission works seamlessly. In order to perform routing between different protocols, the adapter holds an internal routing table to control traffic flowing between the networks.

4.2.3 Gateway

The task of the Gateway is to bridge the WHART and the plant automation network, enabling host applications to gain information from the network, and experts are able to analyse the readings sent from the field devices. The Gateway also needs to cache burst data from the field devices. As a network only has one active Gateway it has a fixed, known address *0xF981 0000 02* and the nickname *0xF981*. The method of transmitting the data highly depends on the protocol, and is usually not wireless. Gateways have a unique clock source, which provides the network with a source to synchronise clock signals used for timeslots and superframes. As can be seen in figure 4.1, the Access Points are connected to the Gateway and will get the clock signal from it, before distributing it onwards to the devices. The Gateway can be used for several tasks:

- Converting data from one protocol to another
- A connection point for one or more networks using the same protocol
- Converting data from one format to another

Virtual Gateway

The Virtual Gateway serves as a single point of entry with the Gateway and Network Manager on one side, the Access Points and network devices on the other. The Virtual Gateway sources the time synchronisation packets to the Access Points and provides buffering for different data packets such as publish data messages, event notifications, cached command responses, diagnostics and large data transfers. The Virtual Gateway is also tasked with supporting existing HART commands in cases where the Gateway works as a translator.

4.2.4 Access Points

The Access Point provides access to the wireless network for the Gateway, Network Manager and the Security Manager, making them able to communicate with the field devices. The connection to the Gateway does not have to be a WHART connection. It can also be connected through other

network connections like Wi-Fi or Ethernet. The Access Point supports communication to any device part of its subnet.

4.2.5 Network Manager

The Network Manager has the task of configuring the network and performing scheduling and routing. Although it is possible to have redundant Network Managers, there is only one active Network Manager at any time in the network. The Network Manager has a fixed address (Unique ID = *0xF980 0x000001*; Nickname = *0xF980*) which is known by all the devices in the network. The Network Manager is tasked with scheduling the route packets will take through the network. Additionally it has to monitor how traffic flows, manage and optimise communication resources, making it the most critical part of the network as it determines how the network acts.

4.2.6 Security Manager

The Security Manager is responsible for generating, storing and managing keys used for device authentication and data encryption in the network. The Security Manager provides the Network Manager with join, network and session keys, while the network devices receive a join key. As the Security Manager is part of the Network Manager, there is only one Security Manager in a network. However one Security Manager is able to provide security for several non-related networks. The Security Manager is completely hidden from the Gateway, as communication goes through the Network Manager only.

4.2.7 Handheld Device

Handheld devices can be directly connected to a WHART network by connecting to a field device. The handheld devices are used to control and perform maintenance on the network, as well as installation and control of network devices.

4.3 Communication between the layers

The following sections provide an overview of how communication between the layers are achieved.

4.3.1 Service Primitive types

As the protocol is constructed through a layered approach, the layers within the stack communicate through sets of Application Program Interfaces (APIs) which every layer provides to the layer above, also called Service Primitives (SPs) [6]. There are four message types, namely:

- Request - Upper layer requests data or a service from a lower layer

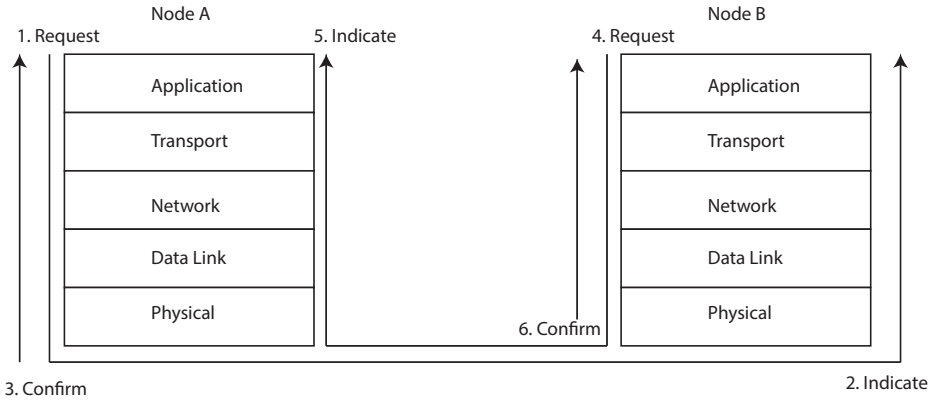

Figure 4.2: Flow between stack layers

- Confirm - The lower layer responds to a request
- Indicate - The lower layer sends a request to the layer above. This usually only happens when a new message has arrived.
- Respond - Upper layer responds to the indicate message from the lower layer.

The messages can further be split into two groups; Message service primitives and Management service primitives. Message SPs are transmissions of messages between nodes and Management SPs deal with the configuration of the stack layers.

The message sequence of a typical transaction can be seen in figure 4.2.

- (1) A transmission is initiated by the Application Layer sending a request to the layer below at node A.
- (2) Once the message is received at node B, an indicate message is sent from the Physical Layer to the Application Layer, notifying it of the transmission.
- (3) Once the Physical Layer has transmitted the message from node A, it returns a confirm up the stack.
- (4) The Application Layer at node B creates a reply and sends a request message to the Physical Layer for transmission.
- (5) Node A receives the message and initiates an indicate message, while the Physical Layer at node B returns a confirm (6).

4.3.2 Layer-specific Service Primitives

Each layer provides a set of service primitives for the layer above and beneath. In the following sections the service primitives are briefly explained, while the management primitives are mentioned at the end of each section.

Physical Layer Service Primitives

The Physical Layer is mainly tasked with packet transmission on the radio, and the service primitives are related to the topic.

ENABLE.request - This Service Primitive is used to set a physical channel to either transmit or receive

ENABLE.confirm - responds to an *Enable.request* from the MAC layer with the state a channel has been set to.

ENABLE.indicate - Notifies the MAC layer of an incoming transmission on the radio.

CCA.request - This service primitive is a request from the MAC layer to check whether the channel is clear to send on or not.

CCA.confirm - This service primitive is a response to a *CCA.request* from the MAC layer.

DATA.request - This service primitive requests transmission of a data packet

DATA.confirm - This service primitive responds to a *Data.request* with a status of the transmission

DATA.indication - This service primitive notifies the MAC layer that a whole packet is arriving on the radio, in contrast to *Enable.indicate* which only notifies of data on the link.

ERROR.indication - This service primitive notifies the MAC layer that an error occured while receiving a data packet.

The layer also provides a set of management services which is related the state of the radio and its power level.

Data Link Layer Service Primitives

The service primitives at the DLL are an extension of the services from the neighbouring layers, but also contains services related to the network.

FLUSH.request - This service primitive requests that a packet stored in the transmit queue is to be deleted. The given packet to be deleted is sent along as a parameter.

FLUSH.confirm - This service primitive responds to *Flush.request* to delete a packet from the transmit queue.

TRANSMIT.request - This service primitive is used to request a packet transmission. Depending on the parameters called with, it defines what address type to use and type of routing.

TRANSMIT.confirm - This service primitive is a response to a previous *Transmit.request* about a transmission.

TRANSMIT.indicate - This service primitive notifies the MAC layer about a successful reception of a packet on the Physical Layer.

DISCONNECT.indicate - This service primitive notifies the Network Layer about a device which has disconnected, allowing the Network Layer to reschedule its routes.

PATH_FAILURE.indicate - This service primitive notifies the Network Layer about a failing link.

ADVERTISE.indicate - This service primitive notifies the Network Layer about a received advertisement message. If the device is not part of the network, the layer can initiate a join process and attempt to synchronise to the network the advertisement message came from.

NEIGHBOUR.indicate - This service primitive notifies about a neighbour discovery which is not formerly recognised as a neighbour. Updates in the routing table can be made accordingly.

RECEIVE.indicate - This service primitive notifies about a received message which destination is not this device. The Network Layer will check its routing table, before sending it onwards on the correct link.

In addition, the Data Link Layer provides a set of management services which is related to managing superframes, links, edges, network ID and join related information.

Network Layer Service Primitives

The Network Layer contains services related to transmission.

TRANSMIT.request - This service primitive is called to transmit a packet.

TRANSMIT.indicate - This service is called when a packet is received.

TRANSMIT.response - This service primitive is used to respond to a *Transmit.indicate*.

TRANSMIT.confirm - This service primitive is used to confirm a *Transmit.request* with a status message.

FLUSH.request - This service is used to delete a packet with the handle sent along as a parameter.

FLUSH.confirm - This service is a response to a *Flush.request*.

In addition, the layer provides a set of management services providing network tasks related to sessions, routes and Time To Live (TTL) of a packet.

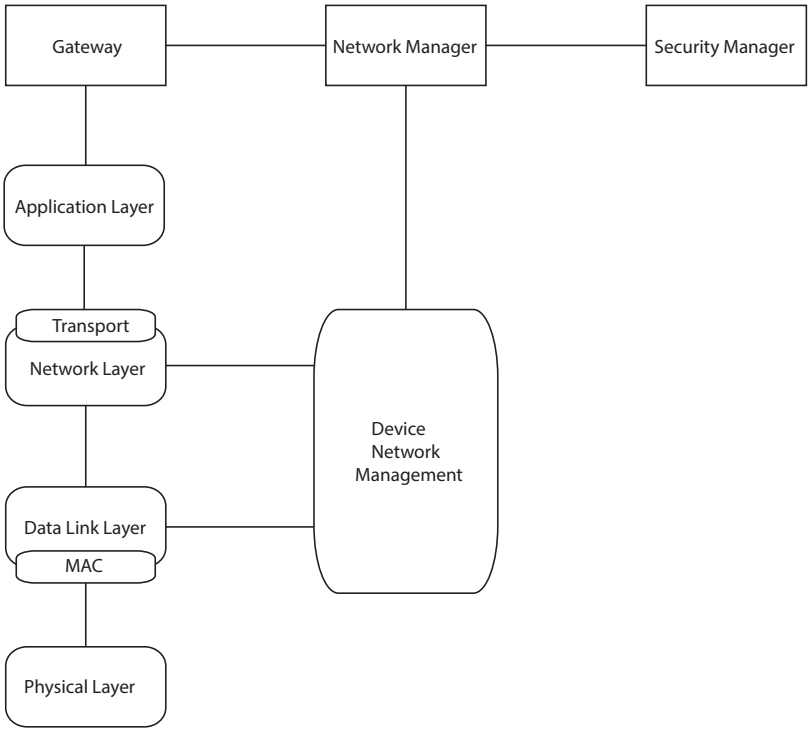

Figure 4.3: WirelessHART communication stack

4.4 WHART Physical Layer

The WirelessHART layered approach is similar to that of the OSI model widely used to describe many communication protocols [17]. The WirelessHART protocol introduces additional features at some of the layers, which we will look at in the following sections.

The Physical Layer is based on the IEEE 802.15.4-2006 2.4 GHz Direct Sequence Spread Spectrum (DSSS)[3] Physical Layer [6]. It deals with every characteristic related to transmission such as signalling method, signal strength and device sensitivity. The radio only operates in the 2400 - 2483,5 MHz ISM band, and is able to send at 250 Kbps. The radio has 16 channels numbering from 11 to 26, split by a 5 MHz gap.

4.4.1 Clear Channel Assessment

Clear Channel Assessment is a technique used in wireless networks to determine if a channel is busy or not. CCA has three different modes for determining a channel as free of traffic:

Mode 0 Energy detection above a specified threshold

Mode 1 802.11 DSSS physical packet signal detection. The receiver is constantly monitoring the channel for a 802.11 preamble.

[3]Mechanism to spread transmissions across a wider range of the spectrum

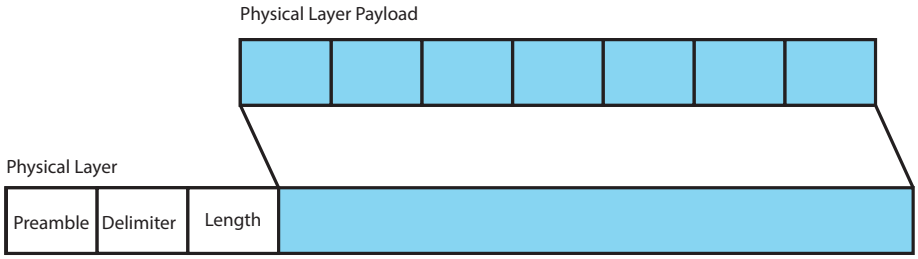

Figure 4.4: Physical PDU structure

Mode 2 A combination of mode 0 and 1.

WirelessHART utilises mode 2 of CCA, determining the channel as clear if no IEEE 802.15.4 traffic is detected.

4.4.2 Frequency Hopping Spread Spectrum in WHART

Frequency Hopping Spread Spectrum (FHSS), also called Channel Hopping, is a technique used in wireless networks to avoid jamming and reduce interference by other devices during transmission. The physical channel is changed for every transmission, ensuring that only the authenticated devices know which channel to use at any time. The standard uses 16 channels numbered from 11 to 26, where channels can be blacklisted if they are already in use or are noise infected. There are two different types that can be used:

Slotted hopping One message per slot only.

Slow hopping A slot can be shared amongst a group of devices, which is based on the CSMA/CA channel access described in section 2.2 on page 13.

Figure 4.6 displays how a network can utilise channel hopping for every new transaction.

4.4.3 Physical Layer Protocol Data Unit

The Physical Layer PDU consists of three values used by the radio to inform about an incoming packet. The three values are:

Preamble - A four bytes sequence used by the radio to inform that a packet is arriving.

Delimiter - Also called *Start of Frame Delimter (SFD)*. Consists of one byte, notifying the radio that the next byte is part of the packet.

Length - Seven bytes containing the length of the data packet, excluding the PPDU.

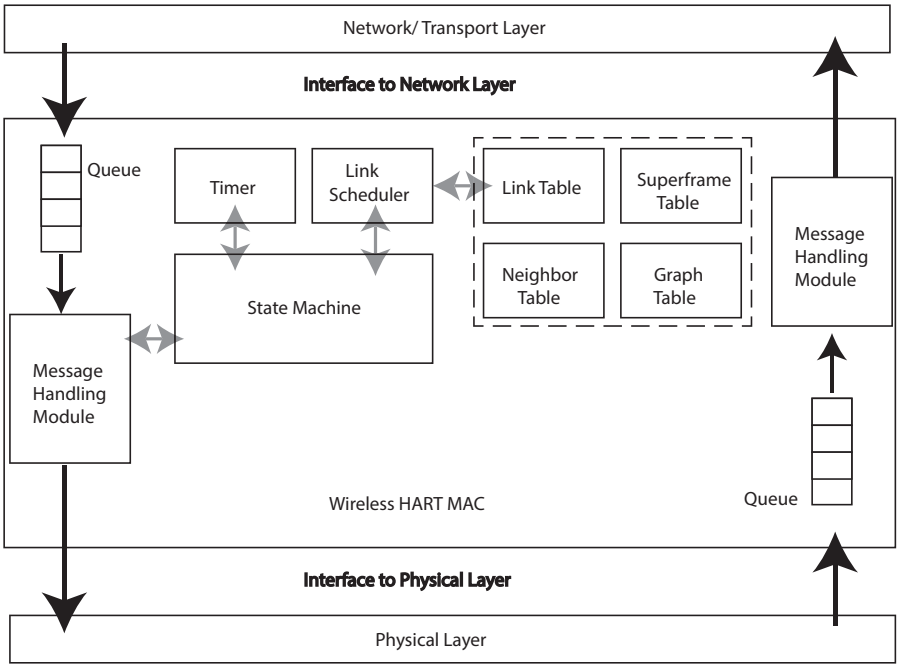

Figure 4.5: Data Link Layer Architecture

4.5 WHART Data Link Layer

The Data Link Layer (DLL) provides mechanisms for time synchronisation and transmission between neighbours. As WHART uses TDMA (See section 2.1 on page 12) to avoid collisions by assigning timeslots (section 4.5.1), it is critical to keep the clocks of all devices in the network synchronised. Every timeslot is 10 ms, meaning the DLL will have to constantly update its clock due to clock drift. To keep track of timeslots, the DLL stores information in superframes (see section 4.5.1). As the DLL has different tasks, they can be split into six main modules seen in figure 4.5 and described in the following sections:

Interfaces

As the WHART protocol uses a layered approach, every layer needs interfaces to be able to access and be accessed by its neighbouring layers. The layer provides services for interacting with the Physical Layer and the Network Layer. The Interfaces are only represented by bold text in figure 4.5.

Timer

The task of the timer module is to keep the clock of the device synchronised to the network time, so that the time slots can be met accurately. As the time slots are of 10 ms, it leaves small room for error.

Communication Tables

WHART is a mesh network, where every node needs to keep a minimum of routing information. The communication table maintains a collection of the node's neighbours and connecting links. This information is provided by the Network Manager, and is used for routing packets through the network. The communication tables are *Link Table*, *Superframe Table*, *Neighbour Table* and *Graph Table* in figure 4.5.

Link Scheduler

The link scheduler calculates the next active timeslot based on information from the superframe (see section 4.5.1) and link tables (see section 4.5.2). As priorities of transactions and links can change, the link scheduler continually needs to recalculate the order.

Message Handling Module

The DLL keeps two separate queues; one for packets passing up to the network layer and one for packets down to the physical layer.

State Machine

The state machine performs the tasks of receiving and sending packets. As the sending and receiving is connected to timing, this module keeps a TDMA state machine as well as *transmit* and *receive engines*[4]. The state machine is responsible for sending and receiving packets during a given timeslot, to adjust the clock when needed and transition from one state to another according to the superframe.

4.5.1 Time Division Multiple Access

TDMA is a mechanism for splitting a shared media between devices as discussed in section 2.1 on page 12. WirelessHART assigns the Network Manager to allocate how the medium access is distributed. The Network Manager creates superframes consisting of a variable number of timeslots, and schedules in what order the superframes are allowed to access the medium as can seen in figure 4.6.

Timeslot

A timeslot is a period of 10 ms during which a device can communicate with the network. The Network Manager is responsible for allocating timeslots to devices. Timeslots are assigned to one device and specifically for one link. During that timeslot a device can send packets on the link, and the neighbouring device will be set to receive mode. The Network Manager orders timeslots in superframes (section 4.5.1) distributed among nodes.

[4]The mechanisms responsible for transmitting and receiving packages

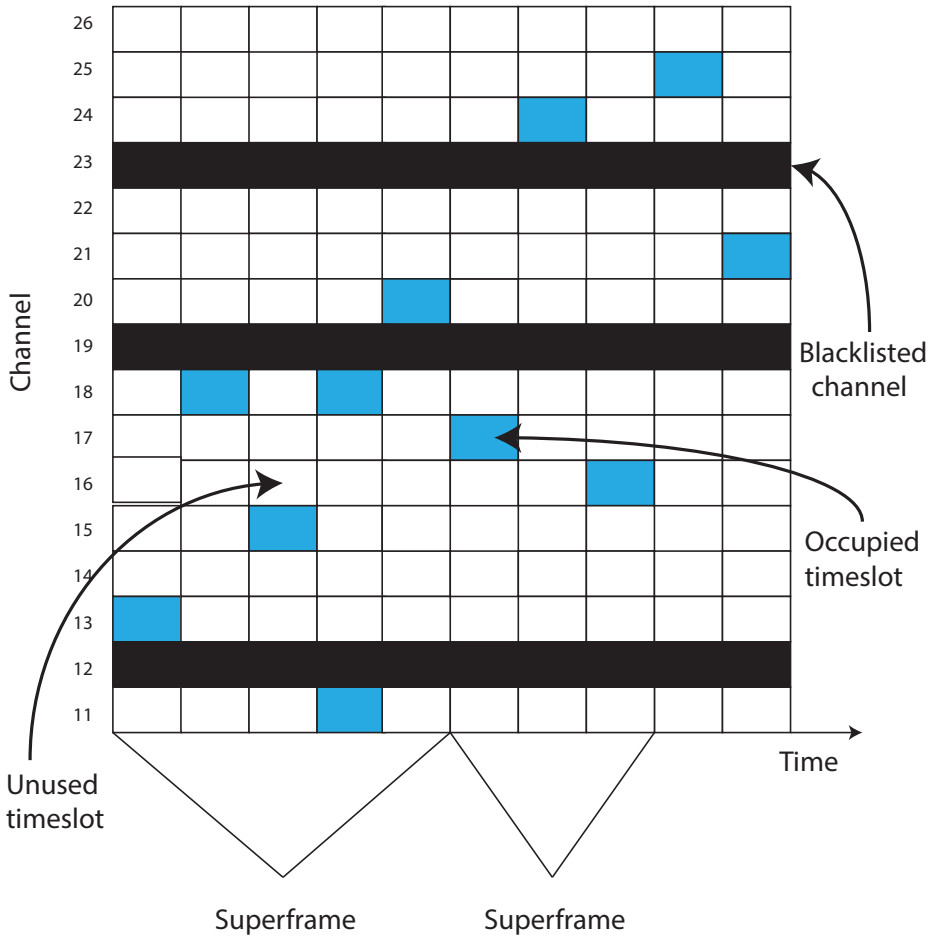

Figure 4.6: TDMA with assigned superframes

Superframe

Superframe is used in combination with TDMA (see section 2.1) for ordering timeslots and assigning a specified time for sending or receiving at devices as illustrated on figure 4.6. It is possible to have multiple superframes, where each one determines a specific sequence of time slots. Every superframe start from ASN 0 (Section 4.8.1) and has a length of the number of timeslots. The superframe will repeat itself once it reaches the end. The Network Manager assigns timeslots to devices according to needs, and specifies whether it is a send or receive timeslot. As timeslots are only one way, if a neighbouring pair of devices want to communicate they will require one timeslot each for transmission.

4.5.2 Links

How communication between two devices during a timeslot happens, is determined by links. Every occupied timeslot is assigned to a link, and depending on the type of link there is a sender and a number of recipients. There are four types of links, briefly discussed in the following subsections:

Normal Link

Normal link is the common link connecting two devices, where there is a sender and a receiver. The link is used by two devices to communicate between each other once they both are fully connected to the network. The transmission packets are usually Data DLPDU's discussed in section 4.5.6, followed by an ACK DLPDU if there is enough time left in the timeslot.

Broadcast Link

A broadcast link is a link which is assigned to a sending device, and every device within range is considered as a recipient device. As no one device is the receiver, the receiving address is set to *0xFFFF*. Opposed to messages sent on a normal link, broadcast link messages are not acknowledged by the receiving devices.

Join Link

A join link is used during the join process of a new node, and can be considered as a partly integrated link only. The link is associated with the joining device, and depending on which way the messages flow, the device can be either the sender or the receiver. The join link is after a new device intercepts an advertisement message (see section 4.5.6) and constructs a join request from information contained within the advertisement message. The join request message is sent through a neighbouring node as shown in figure 4.20 on page 56. Once received and processed, the Network Manager schedules a join link for the joining device.

Discovery Link

A discovery link is used to maintain and service a network. The link is used for nodes to discover new neighbours and to maintain connectivity with already existing neighbours. After a set amount of time[5], a device will send a keep-alive packet (see section 4.5.6) to the neighbouring device it has not communicated with for the longest amount of time. Discovery links are downgraded by other links, and will not be used if another link can be scheduled during that timeslot. If a device is not scheduled to transmit during a discovery timeslot, it will still listen to record traffic between its neighbours.

4.5.3 Link Scheduling

All devices maintain a link scheduling function to identify the next slot within an active superframe. When a node *services* a timeslot, it either listens for a new packet or propagates a packet onward. To identify the next link within a superframe, the calculation shown in 4.1 is performed.

$$SuperframeSlot = (AbsoluteSlotNumber)\%Superframe.NumSlots \quad (4.1)$$

Determining the next link is straightforward, but is complicated by events requiring frequent recalculation of assigned slots. Events affecting the scheduling are transaction priorities, modification of links, enabling and disabling of superframes or failed transactions of high priority.

4.5.4 Neighbour Table

Every device keeps and maintains a list of its neighbours; both the ones communicating with and the ones discovered when overhearing communications. The neighbour table is critical for communication and maintaining a mesh network, as each link can either be connected to one neighbour or a broadcast link, or to a graph referencing multiple neighbours. The table keeps information relevant for the device to perform communication successfully, such as:

General neighbour identity information Contains the neighbour's unique ID, 2-byte nickname address and if the device is a time source.

Performance and historical statistics contains statistics relevant for communication such as Received Signal Level (RSL), a timestamp for last communication with neighbour and the number of packets transmitted and received.

Shared slot parameters contains information needed to support the back-off algorithm.

[5] default 30 seconds

4.5.5 The Sub-layers

Several of the functions performed by the Data Link Layer are executed in its two sub-layers introduced below.

Logical Link Layer Control

The LLC sub-layer is assigned the task of creating the different DLPDU packet types, and populate the header fields accordingly as discussed in section 4.8. The layer is also tasked with security and error-correcting at the Data Link Layer, by generating the Message Integrity Code (section 4.5.7) and the Cyclic Redundancy Check (section 8.1).

Medium Access Control

The MAC sub-layer manages slot synchronisation, identifies slots to be serviced, and handles listening for packets sent from neighbours as well as sending DLPDU's. An overview of the structures kept in the MAC are presented in table 4.1. The main priority of the sub-layer is to propagate packets put in the device's buffer, followed by receiving DLPDU's from its neighbours.

Module	**Presented in section**
Neighbour table	4.5.4
Superframes	4.5.1
Links	4.5.2
Link Scheduler	4.5.3
State Machine	4.5

Table 4.1: MAC overview

4.5.6 Data Protocol Data Unit types

The standard specifies different types of DLPDU's that can be sent between devices. Depending on the type, the packet contains different information and are processed accordingly. The type is determined by the value set in the *Packet Type* field in the *DLPDU specifier* seen in figure 4.7. Below we briefly introduce the different packet types.

Data

Data DLPDU is the standard packet types sent within the network. Data packets contain network and device data on route to their destination. 111_2 in the Packet Type field i the DLPDU specifier determines the packet as a data DLPDU.

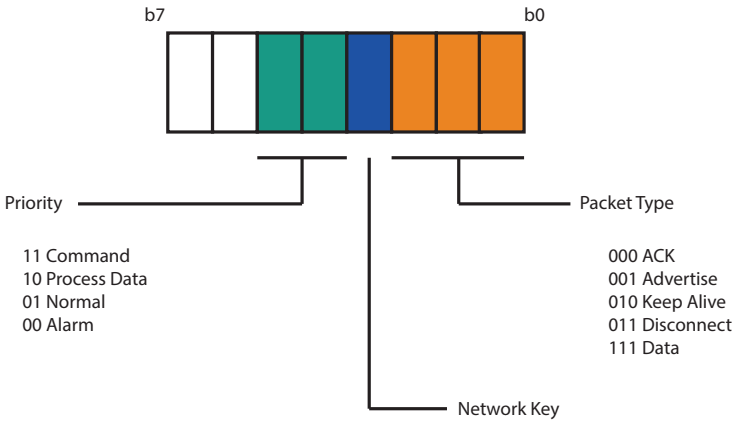

Figure 4.7: DLPDU specifier

Keep-Alive

Keep-Alive DLPDU is used for managing and servicing the network. As the device clocks are subject to clock drift, they constantly need to be adjusted as discussed in section 4.8. Keep-Alive packets are sent between neighbouring devices at regular intervals for maintaining time synchronisation. 010_2 in the Packet Type determines the Keep-Alive DLPDU.

Advertise

Advertise DLPDU are data packets sent on the shared medium to provide neighbouring devices with information needed to synchronse with the network and initiate a join process. The Advertise DLPDU contains information such as superframes, ASN of the network, join control, the security level supported, the channel map array and the Graph ID. The joining device investigates the superframes and uses the join links until it is assigned new links by the network manager. 001_2 in the Packet Type determines a packet as an Advertise.

Disconnect

Devices will from time to time leave a network, sometimes in an orderly fashion and sometimes not. Disconnect DLPDU is sent from a device to its neighbour when it is about to leave the network. The Disconnect packet has no payload, only a MIC calculated by the Network Key. The packet type is set to 011_2.

Acknowledge

Acknowledge DLPDU is used as a response to a sending device to inform that a Data DLPDU has been successfully received on the Data Link Layer. The ACK DLPDU is determined with 000_2 in the packet type field in the DLPDU specifier. The ACK generates its MIC using the same key used in

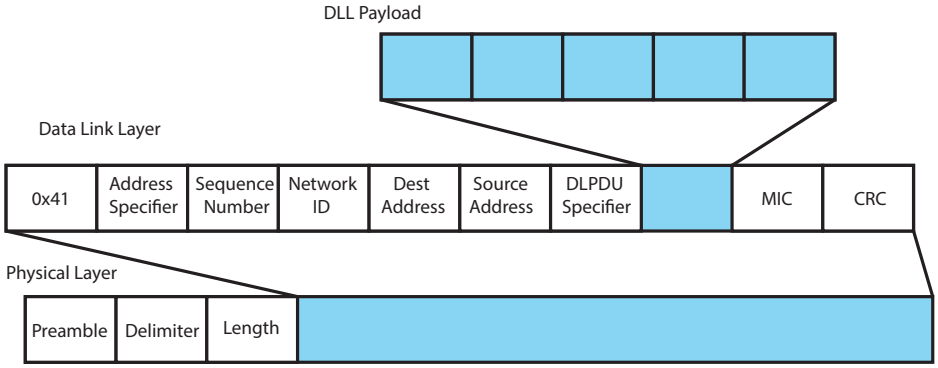

Figure 4.8: Data Link Layer Protocol Unit structure

the received DLPDU. An ACK contains a response code determining if the packet was successfully received or not:

0 - Successfully received

61 - Error due to no buffer space available

62 - Error, no alarm/event buffers available

63 - Error, too low priority

Packets of type Keep-Alive, Advertise and Disconnect addressed to a device are responded with an ACK. ACK packets also contain a Time Adjustment Field, containing the difference between expected time of reception and actual reception. The field is used for time synchronising.

4.5.7 Data Link Layer Protocol Data Unit

The Data Link Layer Protocol Data Unit (DLPDU) is the packet sent between the Data Link Layer and the Physical Layer when transmitting and receiving packets. The DLPDU contains information necessary for neighbour to neighbour transmissions as well as information for ensuring error free transmission.

0x41

The WHART standard requires the first byte to be set to the value 0x41. No further description of the use is provided.

Address Specifier

This field is used to determine which addressing mode is used for the current packet. EUI-64 addressing mode are used during the join process, while nickname addresses are used once the source and destination devices are both part of the network.

Bytes	Value
1	0x41
1	Address Specifier
1	Sequence Number
2	Network ID
2/8	Destination Address
2/8	Source Address
1	DLPDU Specifier
-	DLL payload
4	Message Integrity Code (MIC)
2	Cyclic Redundancy Check (CRC16)

Table 4.2: DLPDU structure

Network ID

The field is used for storing the 2 byte network ID of the network the packet belongs to.

Destination and Source Address

Destination and source address contains the addresses for the neighbouring pair during this transmission. The values can be either 2 or 8-bytes long, depending on the current state of the devices. If one device is in the joining process, 8-byte addresses are used. If not, the 2 byte nicknames are utilised.

DLPDU Specifier

The DLPDU specifier seen on figure 4.7 provides information about the DLPDU. Bit 0-2 identifies the type of DLPDU contained in the packet, further described in section 4.5.6. Bit 3 says whether the network key or join key is utilised, while bit 4-5 determines the priority of the packet. Bits 6-7 are not used.

DLL Payload

DLL payload is the data sent as well as the headers from the layers above.

Message Integrity Code

4 bytes are used to provide a message integrity code of the DLPDU. The MIC is used for authentication, and no packets will be processed unless they are authenticated first.

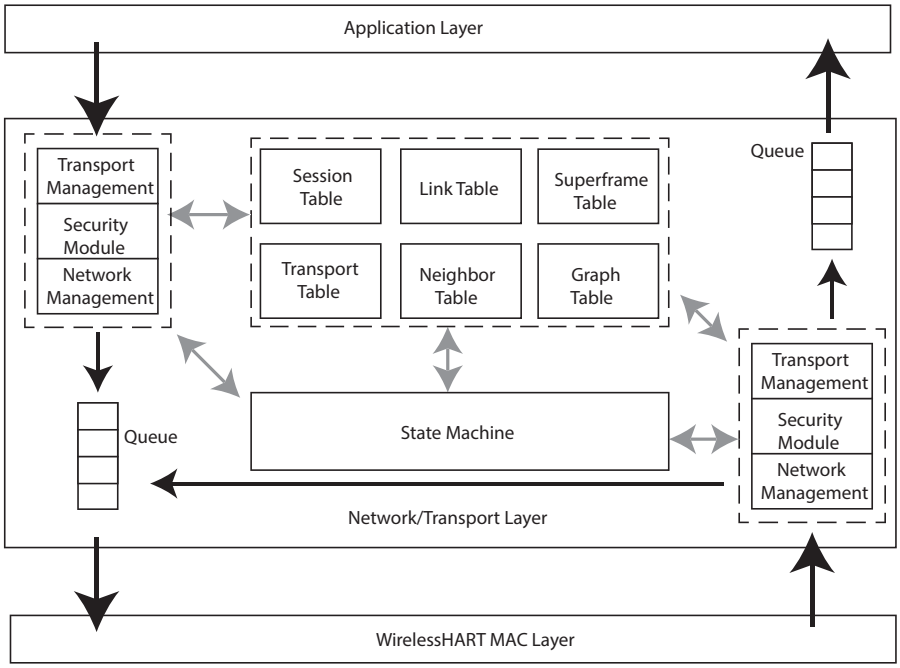

Figure 4.9: Network Layer architecture

Cyclic Redundancy Check

Cyclic Redundancy Check (CRC) is a value generated to make certain that an error-free transmission has occurred. An algorithm generating a check value of the DLL payload is executed, and the value is stored in the CRC field.

4.6 WHART Network and Transport Layer

In the standard, the Network Layer and the Transport Layer are closely tied as can be seen in figure 4.3. The two layers work together to provide secure end-to-end communication for packet transmission. While the Data Link Layer deals with connections between neighbours, these layers provide the routing needed to take a packet from the sender to the receiver.

4.6.1 Routing

In WirelessHART there are three different ways of routing a packet from the source to a sink, namely source, graph and superframe routing. The two first are introduced in section 2.1.5. The type of routing used depends on who the communicating devices are. Communication can go between two devices or between a device and the Gateway.

Source Routing

In source routing the whole path to the destination are enclosed in the header of the packet. Each device on the path simply checks the header, and forwards the packet on the correct link. Source routing is mostly used by the Network Manager and Gateway, as they know the network topology and are able to construct the whole path. In the event a device wants to use source routing, the Network Manager must first provide it with the whole path by *Command 976 Write Source-Route*. The path is associated with the destination address through *Command 974 Write Route.*

Graph Routing

In graph routing, a message is forwarded on the links according to the 2 bytes graph ID field in the NPDU header. The graph ID is a directional graph constructed by the Network Manager, containing neighbours and links which are on the path to the destination device. Every device will choose the first available link matching an edge in the graph, and forward the message. As the graph is directional, delivery is guaranteed as no loops are present.

Superframe Routing

Superframe routing is a special case of graph routing special to WirelessHART. Instead of using the graph ID field, devices use the superframe ID in the message for routing. The message follows a superframe, and every device in the superframe forwards the message regardless of its neighbours. In order to determine if superframe or graph routing is used, superframe routing corresponds to values between zero and 255 in the graph ID field. Values over 255 determine that graph routing is applied.

4.6.2 Network Layer Protocol Data Unit

The Network Layer PDU contains information used for successful end-to-end transmissions, as well as security functionality provided by the security sub-layer.

Control Field

The Network Control Field seen in figure 4.11 contains information on how the rest of the fields in the header should be read. Bits 0-2 indicate if the expanded routing information field contains various information seen on figure 4.10 on the next page. If set, the expanded routing field contains one or two 8-bytes long routes and a 2-byte proxy route. Bits 3-5 are reserved according to the standard, while bits 6-7 determine if the source and destination fields contain EUI-64 addresses.

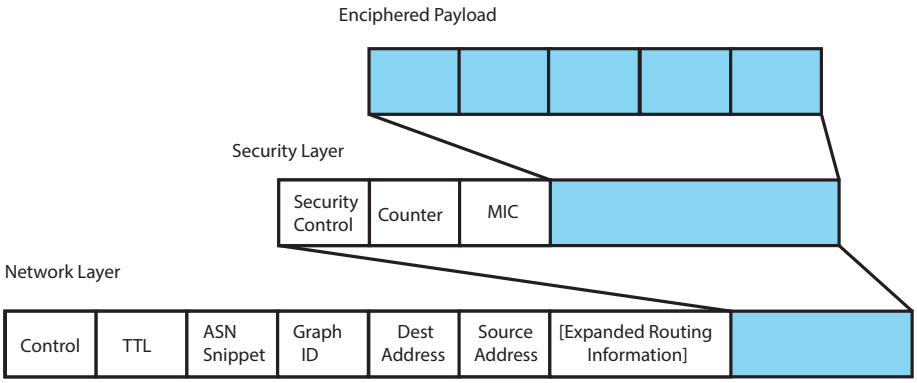

Figure 4.10: Network Layer Protocol Data Unit structure

Byte	Value
1	Control Field
1	TTL counter
2	ASN snippet
2	Graph ID
2/8	Destination Address
2/8	Source Address
0/2/10/18	Expanded Routing Information
1	Security Control
13	Nounce Counter
4	MIC
-	Enciphered Payload

Table 4.3: NPDU structure

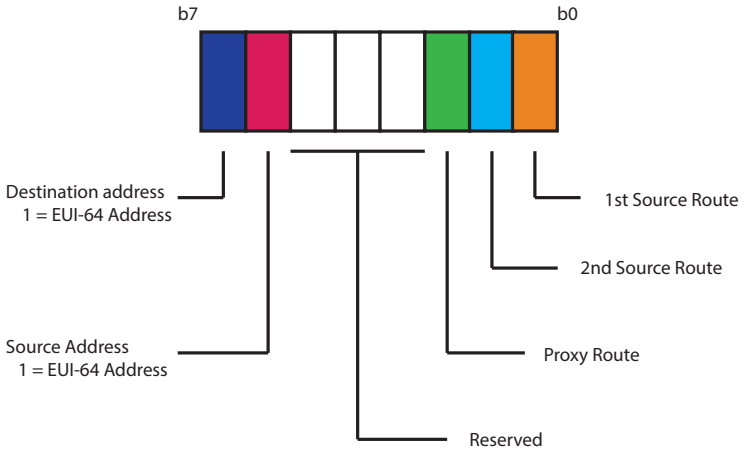

Figure 4.11: The Network Control Byte

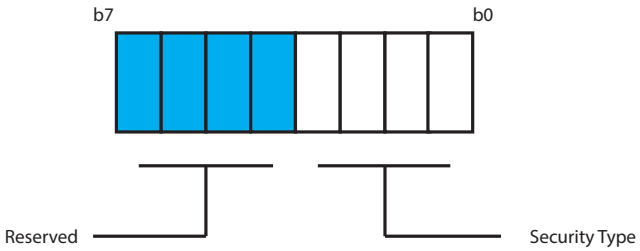

Figure 4.12: The Security Control Byte

Time To Live

Time To Live (TTL) is a hop counter which determines how many more hops the packet can make before being dropped. When the packet is received at the Network Layer, the destination address is checked against the address of the current device. If it is a match, it is further handled. If not, the counter is decremented and the packet is forwarded on a link towards its destination.

ASN Snippet

Contains the two least significant bytes of the Absolute Slot Number (see section 4.8.1 on page 48), and is used to perform diagnostics on the network to determine network efficiency. The packet's ASN is compared to the current ASN to determine the age of the packet. If the value is greater than maxPacketAge, the packet is discarded as too old.

Graph ID

The graph ID contains a set of possible nodes to forward the packet to for reaching the destination. The field is used for routing the packet to the destination, and can be used in combination with any other routing information avaliable in the header.

Destination and Source Address

These fields contain the values for the original sender and the final destination, either which can be two or eight bytes long. These values are different from the similar fields in the DLPDU, as these fields are not modified during transmission.

4.6.3 Security Layer Protocol Data Unit

The Security Layer header is used for ensuring that the payload is not modified or accessed by unauthenticated personnel by providing integrity, encryption and sender reliability.

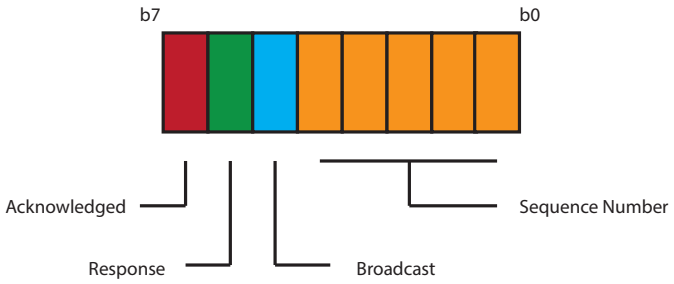

Figure 4.13: Transport byte

Security Control Byte

The first byte of the Security Protocol Data Unit (SPDU) is used to provide information about the security provided as can be seen in figure 4.12. Bit zero to three indicates what security is used to encrypt the payload, while bits four to seven is reserved and not used. The security employed is determined by a type code:

0 - Session Keyed

1 - Join Keyed

2 - Handheld Keyed

Four bits are reserved for three values in order to be able to expand the security functionality in the future.

Counter

The nounce counter used for running the encryption algorithm defined in the Security Control Byte.

Message Integrity Check

MIC provides the SPDU with integrity and authenticity ensurance. It is similar to a Message Authentication Code, except that the generation algorithm does not use a secret key. The algorithm generates a value and stores it in the MIC field. The value is generated again at the receiving end and compared.

4.6.4 Transport Layer Protocol Data Unit

The Transport Layer is responsible for ensuring successful transmission of packets across multi-hop connections. TPDU supports both acknowledged packets like data transmissions and un-acknowledged transactions like broadcast or advertisement depending on the values set in the transport byte.

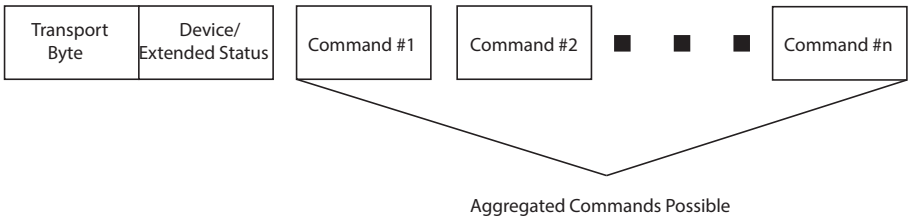

Figure 4.14: Transport Layer

Transport Byte

This byte contains information used to ensure end to end delivery of a packet, as can be seen in figure 4.13. Bit zero to four contains the sequence number used to order and track packets. The field is set differently depending if the traffic is acknowledged or not, or if it is a response packet. Bit five indicates that a broadcast message is used. Bit six is set when the NPDU is a response packet, in which case the payload should only contain command responses. Bit seven is set when the package is to be acknowledged, and is used to make sure that acknowledgements are created and sent.

Device status and Extended Device Status

This field contains the status of the device. The standard does not provide further information on this field, and refers to to HCF Enumeration table 17[6].

HART Commands

The transport layer is tasked with appending the HART commands to the final packet to be transmitted. With a few exceptions, several commands can be aggregated and sent within one packet seen in figure 4.14. This is useful when reading device configurations, or when several commands have to be sent between two devices in sequence such as the join process.

4.7 WHART Application Layer

The Application Layer is the upper layer and provides communication between the network and external applications used for analysing the network. The communication between Gateway and devices are command based, and the Application Layer is responsible for extracting command number from a packet, the arguments enclosed and act accordingly seen in figure 4.15. The layer holds a *queue* for packets arriving from the Transport Layer, which hands them over to the *Parser*. A *Command Processor* communicates with a *Command Handler Module* to build command packets put together in the *Assembler* module. Commands are often relying on

[6]http://www.hartcomm2.org/hcf/services_tools/doc_sales.html

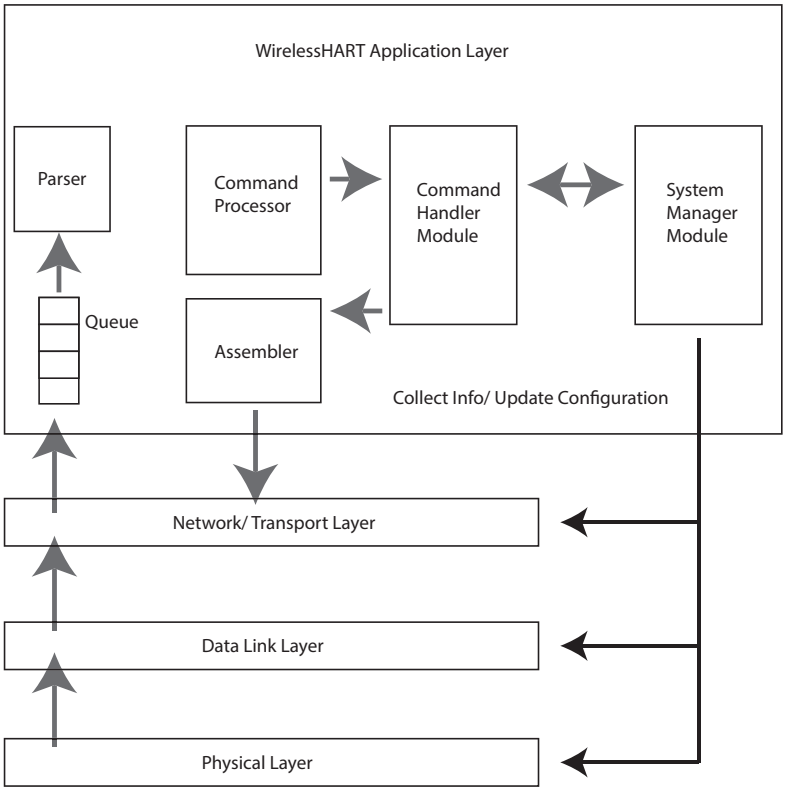

Figure 4.15: Application Layer Architecture

information provided by the other layers, which is requested by the *System Manager Module*. The Application Layer is also where report packets from devices end up, making it possible for external programs to act according to results. The messages passing between layers when requesting commands can be seen in figure 4.2.

4.8 Time Synchronisation

Time synchronisation is critical in a WirelessHART network to enable devices to communicate. The Gateway is the ultimate time source, while the Access Points propagate the time synchronisation in the wireless network. In a network there can be several Access Points providing clock signals, and they all have to be synchronised. The communication between a Gateway and the Access Points is not specified, but by utilising a wired connection the time challenge between the devices can be solved.

4.8.1 Absolute Slot Number

Absolute Slot Number (ASN) is a counter used by the devices to calculate when they are scheduled to transmit or send packets. ASN is determined by the number of timeslots (see section 4.5.1) since the network were created, at which ASN were 0. By knowing the global time when ASN

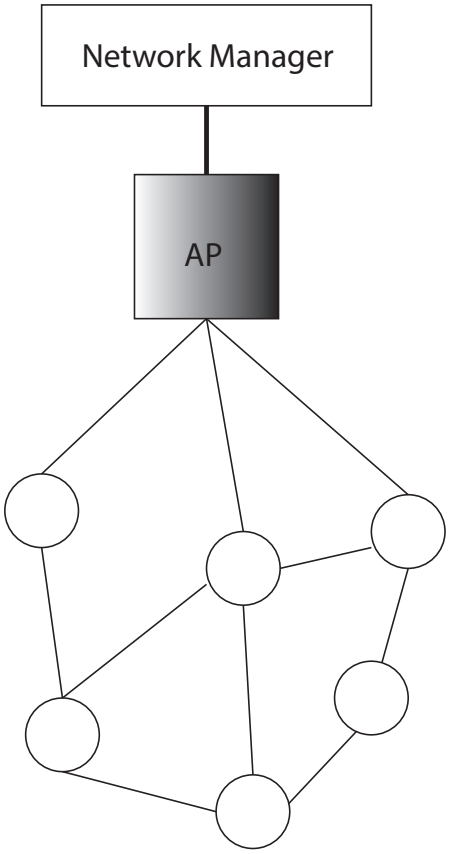

Figure 4.16: Single Access Point with clock

were 0, a device can determine the current time by calculating **number of current ASN * 10 ms**[7]. The device achieves this by receiving *Command 793 Write UTC Time Mapping* from the Network Manager. This command informs the device what time it was when the network were created, and should help the node synchronise to the network.

4.8.2 Providing clock signals

In WirelessHART there are multiple ways to provide time synchronisation throughout the network. Access Points are usually given the task of propagating the clock signals throughout the network. As a network has N[8] Access Points, there can be multiple clock providers. The most effective number of time synchronisation providers depends largely on the size of the network. Figure 4.16 displays a single Access Point providing time synchronisation to the whole network.

In figure 4.17 we have one Access Point connected to the Network Manager which provides time synchronisation, and another AP which does not. As several Access Points can be favourable to provide redundancy and

[7] every timeslot is 10 ms
[8] N = 1,2,..

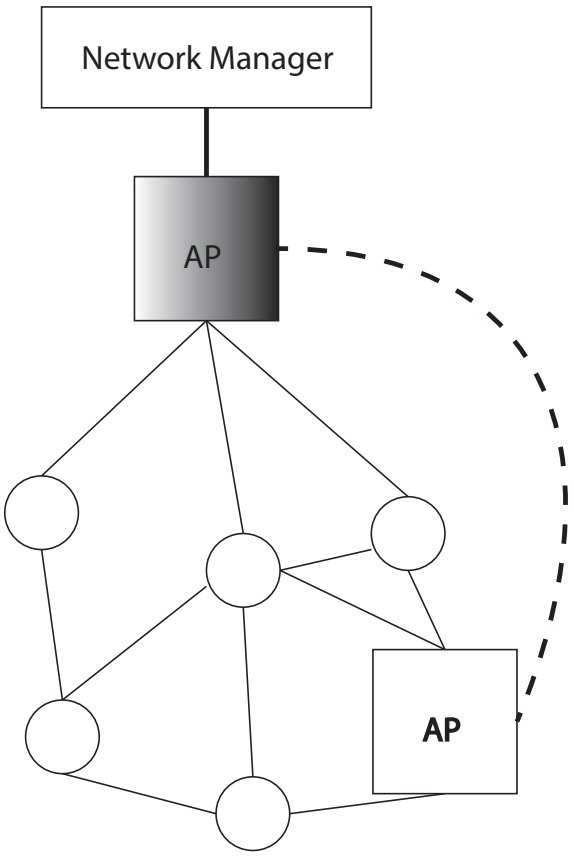

Figure 4.17: Access Point without clock

optimal paths, it might not be necessary for each one to propagate a clock signal.

Figure 4.18 shows a larger network being serviced by two Access Points providing clock signals. As a network grows larger, it is necessary to have multiple time providers in order to keep the network synchronised and prevent clock drift from affecting the efficiency of a network.

4.8.3 Slot Timing

Every communication session between nodes follows the assigned timeslot as discussed in section 4.5.1. Clocks can drift due to the variance in crystal oscillation[9]. This causes the devices to consider timeslots differently, resulting in shifting and failing transmissions within a timeslot. To keep the clock of two separate devices aligned, a transmitting device will transmit its packet when it perceives its timeslot to start. The device scheduled as the recipient, will enter receive mode when it estimates the timeslot to begin. From the timeslot start, a *TsRxOffset* value is calculated before the receive window(*TsRxWait*) is allowed to drift while trying to resynchronise the devices. Once a message is detected after the end of the Physical

[9]Caused by temperature, ageing and other effects.

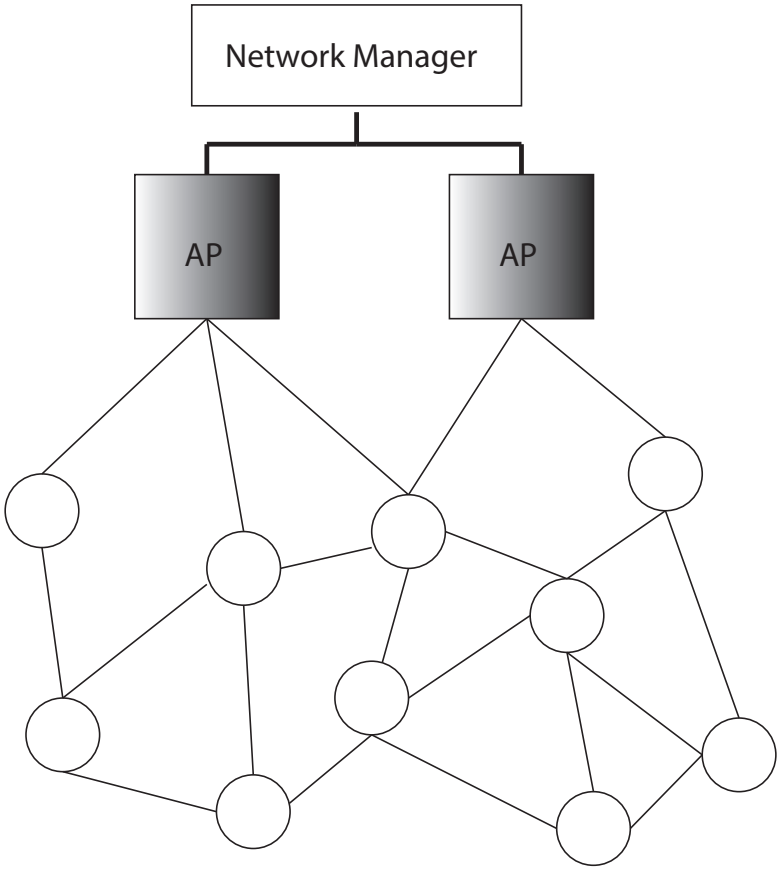

Figure 4.18: Multiple Access Points with clocks

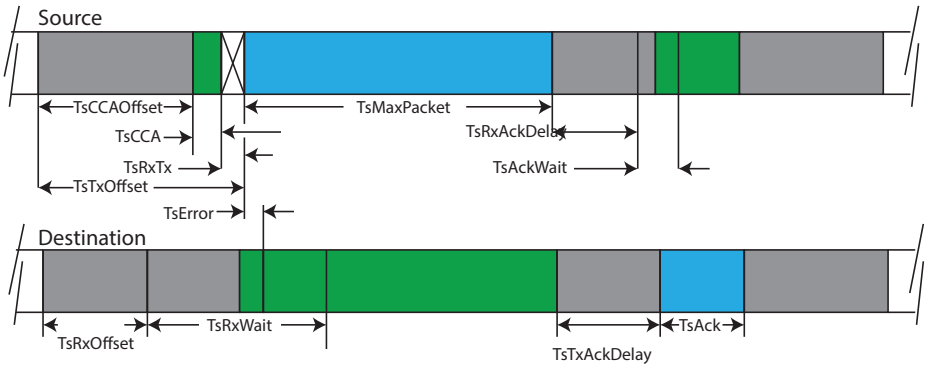

Figure 4.19: Slot timing

Layer Delimiter, the value *TsError* is calculated determining the difference between the device's calculated start time and the real start time of the packet. If the packet had a destination address, an *ACK* is generated and returned, allowing the source address to align its time window according to the destination. The values defined in figure 4.19 are listed below:

TsTxOffset Defines the start of the slot to the start of preamble transmission. (see section 4.4)

TsRxOffset Start of slot to when transceiver should be listening

TsRxWait The minimum amount of time a transceiver has to wait before a message appears. The value determines the clock drift that can be tolerated while still being able to communicate.

TsError The time difference between when the message was sent and when the receiver thought the message was sent. Determines how much the clock has drifted.

TsMaxPacket The maximum amount of time it will take to transmit the largest packet. This includes the *preamble*, *delimiter*, *length* and *DLPDU*.

TsTxAckDelay The time between the end of a message until an *ACK* is generated[10].

TsRxAckDelay From end of transmitting a message to when the transmitting device are supposed to listen for an *ACK*.

TsAckWait The minimum amount of time to wait before an *ACK* can be expected.

TsAck The time it takes to transmit an *ACK*.

TsCCAOffset The amount of time from start of slot to *CCA* begins.

TsCCA The time it takes to perform *CCA*.

[10]Only applies to messages which are not of type broadcast

TsRxTx Amount of time it takes to switch between the two radio states[11].

The Source device

The transaction starts by the source device beginning to transmit the start of the message (SOM) at *TsTxOffset*. *SOM* starts when the Physical Layer Delimiter has been received. Before the message can be sent however, the device performs a *TsCCA* to check if the channel is free of other transmissions. If the channel is free, the transceiver mode is changed to transmit (*TsRxTX*) and the packet is transmitted.

Once the message has been transmitted, the device will change its transceiver to receive mode and wait for an ACK message *TsRxAckDelay* after the packet was transmitted. It will continue to listen for *TsTxWait*.

The Destination device

The destination device will enter receive mode and listen for incoming traffic by *TsRxOffset* during the start of an assigned timeslot. The device will listen for SOM for the duration of *TsRxWait*, and if a message is detected it is received and validated [12].

If the message is validated, the device will inspect the destination address and acknowledge the message[13]. The acknowledgement is done by changing mode to transmit and transmitting an ACK exactly *TsTxAckDelay* after the end of the packet was received.

4.8.4 Keep Alive Interval

As two neighbouring devices synchronise during regular transmissions, they are synchronised as long as communication is upheld on a regular basis. However, if they do not communicate for some time, Keep-Alive packets are exchanged to keep synchronisation[14]. As discussed in section 4.8.3, the communication between two devices rely on different values coinciding. The values can be calculated by the following equations:

$$y + TsRxOffset \leq TsTxOffset \tag{4.2}$$

$$y + TsTxOffset + tHead \leq TsRxOffset + TsRxWait \tag{4.3}$$

Equation 4.2 determines that the destination clock cannot be shifted a larger value than y, in order for the receiving device to start listening before the sender's preamble is started in order to receive the packet successfully. Equation 4.3 determines that the clock cannot be shifted more than y, in

[11] From transmit to receive or vice versa

[12] Messages that cannot be validated will not be handled and acknowledged

[13] Only if it is not a broadcast message

[14] a packet once every 30 seconds is sufficient

order for the destination to receive the physical header before the timeslot is over.

4.9 Join process

The following sections explain the join process of a device in detail.

4.9.1 Overview

When a device wants to join an existing network a communication process between the device, its neighbours and the Network Manager is started.

Initial Device Provisioning

To be able to join a network, a device requires two values. The first is the network ID, which is the unique ID of the network the device attempts to join. The second piece of information is the Join Key, which can be considered the password required by the device to join the network. Once these data are in place, the device can attempt to join the network.

Listening for Network Traffic

Once the node has started the join process, the device engages search mode through its data link. The search mode attempts to synchronise the node with the network so that the node can receive advertise packets. The node will identify its neighbours, both the ones that are advertising and the ones that are not. The device gathers statistics as average signal strength to determine which node it will send the join request through. The neighbour picked is usually the one with proper *Receive Signal Level* and lowest join priority. The signal level is to ensure that the connection between the nodes are not broken. The join priority is a value determined by various factors such as battery power, link load, capacity etc. The lower join priority, the better. The joining node determines the best suited neighbour according to these factors.

Presenting Credentials

Once the device is synchronised with the network, it generates a join request which is sent to the Network Manager. The join request packet contains information the Network Manager needs in order to determine if it should allow further communication with the device. The join request contains:

- The identity of the device
- The long tag which is the MAC address of the node
- The detected neighbouring devices

- The device's Join Key, which is its password into the network. All devices are pre-configured with a Join Key.

The Join Key is used for communication between the device and the Network Manager until the device receives Session Key and Network Key from the Network Manager. As the Join Key is a password into the network, further communication between them means that the Join Key has been authenticated. If the Network Manager decides on further communication, it sends a Session Key used for packet transmissions and a Network Key for Data Link Layer device to device.

For different reasons such as congested links, high join priority or not satisfactory signal level, a device is not able to join the network through the first neighbour it picks. To avoid waiting for a reply that may never come, once the device sends a packet to its picked neighbours it starts a response timer. When the timer expires, a new join request is generated and sent to the next advertising neighbour. This process continues until the device receives a reply or *maxJoinTries* is reached. If *maxJoinTries* is reached, the node goes back to passive search again.

Getting the First Keys

As mentioned in section 4.9.1, there are several factors that needs to be fulfilled in order for a device to join a network. Once the Network Manager receives a join request, it has to use the information contained within the join request packet to establish if certain criteria are met:

- The device has a trusted identity
- The Join Key is a valid password
- The device name and password is satisfactory combined

From the join request packet, the Network Manager has enough information to determine if these criteria are met. If everything is in order, the join request authenticates, the Network Manager assigns a Session Key and a nickname (2-byte address), and send them back to the device along with the Network Key. The NPDU (see section 4.10) is enciphered using the Join Key as this is the only key the device and the Network Manager currently shared. The NPDU is routed back the same way through the parent of the node. Once the joining device receives the NPDU, it sends an acknowledgement back using the new Session and Network Keys.

Device Integration

Now that the device has keys and a network ID, it is partially integrated into the network. Communication between the Network Manager and the device still has to go through the parent as a proxy, utilising the join link. The next step is to fully integrate the device into the network, making it a fully functional node. To achieve this, the device needs at least two time-source parents [9], update the communication tables in the device's parents and transfer the communication from a join link to a regular link.

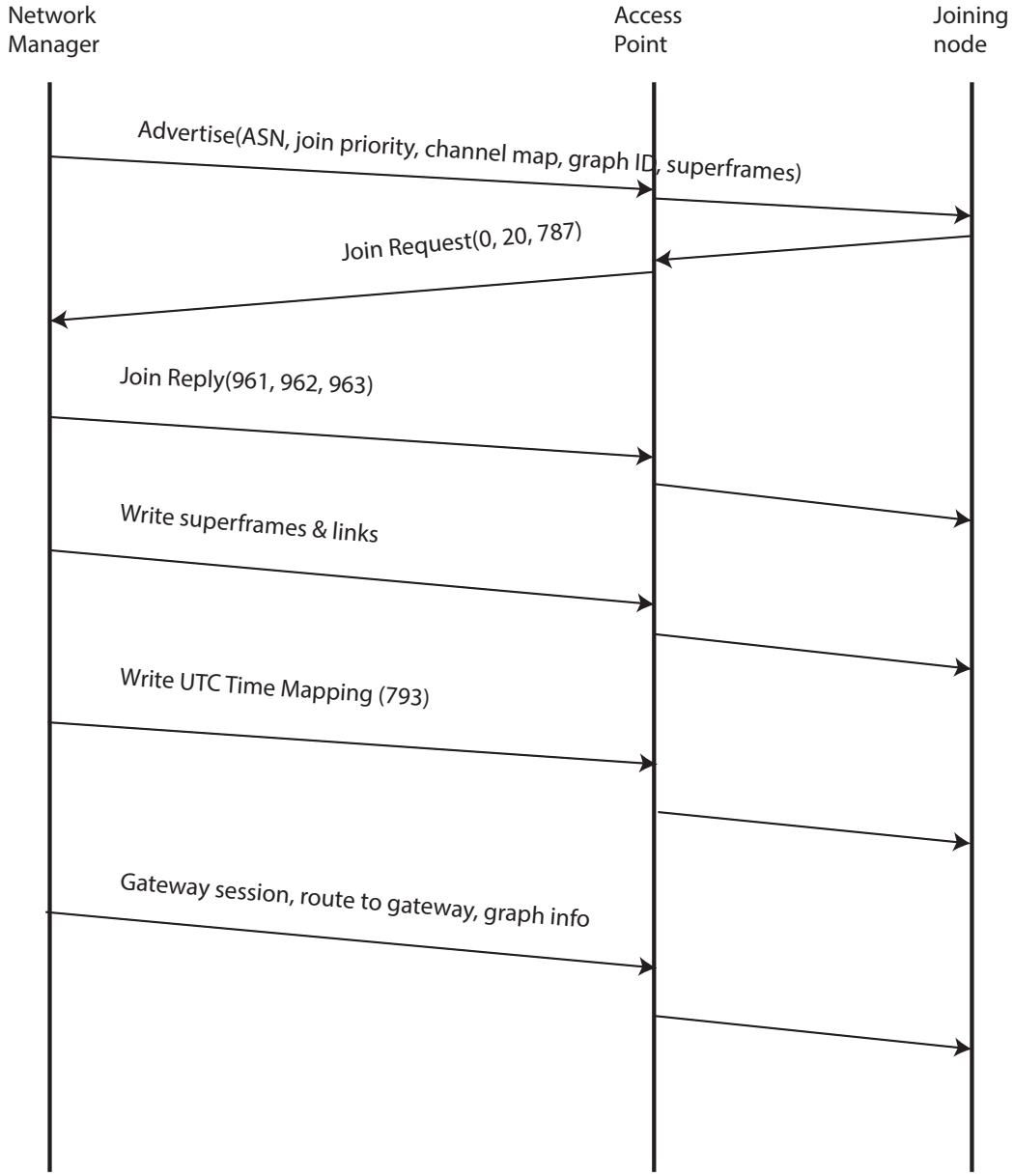

Figure 4.20: Join process

Quarantine

After a device has joined a network, it has no session with the Gateway and is in the state called **Quarantined**. During this state the device can only communicate with the Network Manager and can not publish any data. The device is only allowed to communicate with the Network Manager, but can collect statistics from neighbours and the network. After the device is **Quarantined** it will generate *HealthReports* whenever the *HealthReportTimer*[15] expires. It will continue to do so even after a session with the Gateway is granted. It is up to the Network Manager to decide when the device is given a session to the Gateway. This can happen right after it joins the network or it can be given several minutes.

Becoming Operational

Once a Gateway session[16] is given to the device, the device becomes fully operational. The Gateway will begin filling its data cache for this device. There is usually a surge of activity when a new device connects to the Gateway as can be seen in figure 4.20, so both the Gateway and the device need to obtain sufficient bandwidth to meet the demands. The traffic from the Gateway includes requests from external applications to interact with the device. The bandwidth assigned to the device is used to process data (the publish service) and alarms (the event service).

4.9.2 Network Layer join sequence

Searching

When the Network Layer is in the searching state, it looks for the network and tries to synchronise to it in order to receive DLPDU packets. When the Network Layer receives an advertise packet, it knows that the device has found a neighbour and moves over to the **Got an Advertising Neighbour**. The Network Layer join process can be seen on figure 4.21.

Got an Advertising Neighbor

In this state the Network Layer starts a timer called *AdWaitTimeout* and continues to wait for more advertise packets. Once a sufficient amount of advertise packets has been received, the state shifts to **Requesting Admission**. If the timer runs out, the device tries to connect through the next neighbour as discussed earlier.

Requesting Admission

In this state, the device sends a join request to the Network Manager, and starts a clock called *JoinRspTimer*. *JoinRspTimer* will run until one of the followings incidents happen:

[15] Default value is set to 15 minutes.

[16] The device can be accessed by host applications through the Gateway

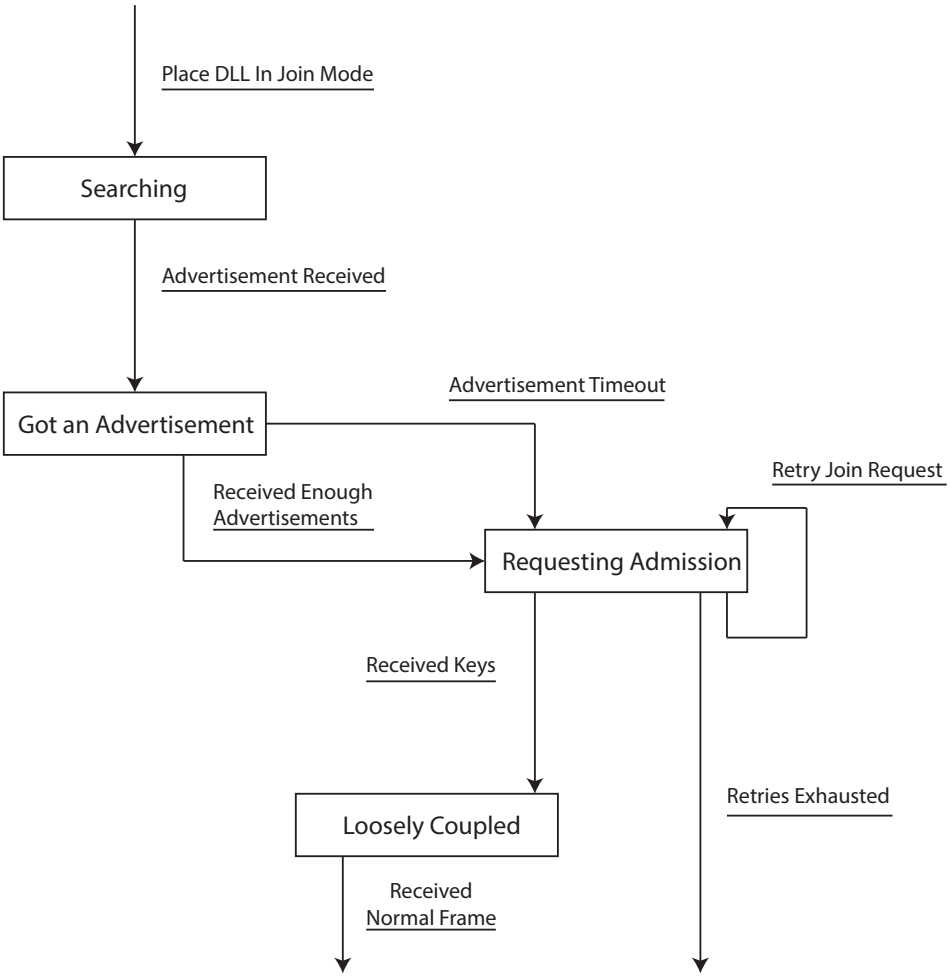

Figure 4.21: Network Layer join procedure

- The Network Manager responds with a Network Key, the Network Manager session and a nickname for the device. If this happens, the device moves onto the **Loosely Coupled** state.
- The *JoinRspTimeout* occurs. If so, a join request is resent and the timeout is started again. This continues until the maximum amount of retries has been reached or the device gets a reply. If no reply occurs, the state machine (see section 4.5) exits with an error.

Loosely Coupled

When the device has received the required information in the **Requesting Admission** state, it can communicate with the Network Manager. However, the communication is only semi-operational as they have to communicate through the shared link and by going through the device's parent. They continue to communicate this way until a normal frame and links have been created for the device. Once this has been ensured, the join process at the Network Layer is complete.

4.9.3 Data Link Layer join sequence

It is the Network Layer that initiates the Data Link Layer join process. The device enters search mode and tries to synchronise its timeslot to the network's slot timing. In order to achieve this, the device starts the **Active Search**.

Active Search

In the **Active Search** mode, the device enters receive mode and listens for packets until the timer *ActiveSearchShedTime* expires. In order to find the network, the device often has to scan through multiple channels. The device sets a time period for listening to every channel called *ChannelSearchTime*. Once the time period expires, the next channel is searched. If a packet is found during the search, the state transitions to **Packet Received**. If *ActiveSearchShedTime* expires, the device enters **Passive Search**.

Passive Search

If no packet is found during **Active Search**, **Passive Search** is entered. This mode is a reduced version of the **Active Search**, where the search cycle is reduced and power saving mode is entered between search intervals. This cycle will continue until a packet is received and the device transitions to **Packet Received**, or the device is forced back into **Active Search** by receiving the *Force Join Mode* command.

Packet Received

The device enters this state once during join. During this state, the mission is to synchronise the device timer to the network time. The device will continue to receive network packets, and once a non-ACK packet arrives the timing of that packet is recorded as the device's slot time. Subsequent non-ACK packets will be used to create statistics and align the time slot further. Once the device's time slot is synchronised to the network's, the device is fully synchronised with the network. The device uses packets received to update its tables and advertise packets to limit its search to certain channels. When the slot time is synchronised and the *absolute slot time* is aligned to the network, the **Slot Time Synchronised** state is entered.

Slot Time Synchronized

When this state is entered, the device has sufficient information about the network to only listen at certain slot times. When the device receives packets during these intervals, the timer is adjusted to keep in sync with the network. Before the Data Link Layer can do anything else, it has to wait for a join request from the Network Layer. The Network Layer however, will not generate the join request until it receives an advertise packet. When the DLL receives the advertise packet, it signals the Network Layer by a *Advertise.indicate* packet. The advertise packet contains information on how the device has to configure Join frames and links in order to join the network. When the join request packet is received from the Network Layer, the DLL can enter the final stage.

Join Frame Active

The device has now configured everything it needs, aligned to the shared time slots from the advertise packet and is ready to join the network. In order to join the network, a request has to be sent via a neighbour. The selection method assigns the neighbour with sufficient receive signal and lowest join priority. Once the request is sent, Keep-Alive packets are sent to its neighbours at regular intervals to keep in synchronisation. The reason for this is to keep an updated overview of its neighbours in case a different node has to be chosen for sending a join request.

4.10 Chapter summary

This chapter has presented the WirelessHART protocol in depth. An overview of a WirelessHART network and its components has been presented in section 4.3, followed by how communication between the different layers are achieved. The functionality and architecture of each layer have been presented in section 4.4 - 4.7. Time synchronisation and the process of joining a new device into a network are two complex tasks, and are two separate sections are dedicated to explaining these processes on pages 48 and 56.

Part III

Developing a WirelessHART network

Chapter 5

Planning the project

In the following chapter the reader is introduced to the process of determining the project goals. The original project goals are listed in table 5.1, before the final goals are presented as two separate parts: functional requirements are detailed in section 5.2.2 and the non-functional requirements in section 5.2.3.

5.1 The process

When we first started on the master dissertation our project focus were on networking. More specifically, we were set for working on the Network Layer, to implement a routing protocol and test the performance of a network. We had a meeting with the last students that worked on this project before officially taking over. During the meeting we were informed that our current project goals could not be fulfilled until certain hardware issues were addressed.

There were some serious problems with getting the RzRaven nodes synchronised with their assigned timeslots (see section 4.5.1 on page 34). The reason is that the radios have two clocks; an oscillator and a crystal oscillator[1]. It is only possible to get hold of the oscillator, which makes the nodes miss their timeslots over time due to clock drift.

The result is that no node wakes up during its assigned time, making contact between nodes based on the chance that the node will hit its timeslot. To solve the issue, the previous master students did not implement Time Division Multiple Access (see section 2.1 on page 12) but instead set every node to always listen when they were not sending packets. Battery power is spent within a few weeks instead of the intended years. The amount of power used is not acceptable for a sensor network and is a direct violation of the WirelessHART standard. As a consequence, we had to revise our project goals.

After several meetings with our supervisor Stein Gjessing and specialist on micro controllers at Sintef, Niels Aakvaag, it became clear that we had to address the timing issue in order to get anywhere. We looked

[1] The crystal oscillator is far more precise

at different options, including changing the dissertation if no nodes that satisfied the response time requirements defined in the standard could be acquired. After considering different suppliers, we initially decided on obtaining new micro controllers from Atmel of the type ATMEGA128RFA1 Evaluation Kit [4]. Besides not having to rewrite library calls to a different api, these chips are a single chip solution providing an API supporting calls to both clocks. This makes it possible to use the crystal oscillator on the boards for better accuracy.

Obtaining new nodes proved a greater challenge than expected, as Atmel were not able to supply us with new nodes at the moment, which lead us towards obtaining new nodes in a different way.
Our supervisor proposed that we had a meeting with Trygve Harvei at ABB [2], who were the examiner of the previous students and is an expert on the WirelessHART protocol. During the meeting with Trygve and his colleague Waqas Ikram, it became clear that our initial goals had already been well researched through the years since the WirelessHART protocol were released. They proposed a new angle on the project since timing issues and node development were already thoroughly researched. The new project were to focus more on the Network Manager, optimising QoS and the time for a node to join the network. This new angle meant we got proprietary nodes of the type WiMon100 from ABB to remove the hardware- and timing issues. Unfortunately this also meant most of the work done by the previous students could no longer be used. We decided after a conversation with Stein Gjessing that we would change the angle, and we made a new project plan. The plan had to be modified after some problems explained in section 7.4.

5.2 Project Goals

The following section provides an overview of the project goals.

5.2.1 General Goals

We started out with a set of goals we thought would be appropriate based on the future work section of the previous thesis and our own expectations for the project. The goals were general and soon proved to be unrealistic, as can be seen in table 5.1. As the project changed, so did our goals. Our main objectives since the start have been to work on the Network Manager and provide routing functionality. In section 7.4 we discuss how new nodes were obtained. Due to new nodes, some of the project goals were reformed into new goals while some were dropped altogether. Goal OFR01 were reshaped to implement a time synchronisation function. OFR02, OFR07, OFR08 and OFR09 were dropped, as these goals were no longer relevant on the new devices. OFR03 was put on hold as large parts of the program would have to be rewritten before we would get to implement channel hopping and blacklisting. OFR06 was met when obtaining new nodes. OFR04 and OFR05 were migrated to the project's new goals, since routing

ID	Project goal
OFR01	Debug and fix timing issues between Gateway, Access Points and sensor nodes
OFR02	Design and implement Network Abstraction Layer
OFR03	Implement channel hopping and blacklisting at the mac layer
OFR04	Identify appropriate routing algorithm
OFR05	Implement routing algorithm on Network Manager
OFR06	Upgrade hardware to support WirelessHart specifications
OFR07	Set up testbed machine with Realtime Linux 2.6 to satisfy the single-digit latency requirement
OFR08	Port the Data Link Layer to C++ on new chips
OFR09	One-hop downlink to get faster response from nodes

Table 5.1: Original project goals

was still relevant. We decided to determine one set of functional and one set of non-functional requirements for our system described in the following sections. This would provide us with two sets of sub-tasks to be completed in order to get a fully functional network up and running. In tables 5.2 and 5.3 we determine the priority of each goal.

5.2.2 Functional Requirements

Functional requirements deal with specific functionality a system should be able to provide, often in the form of what a system must be able to do. Because we are developing a WirelessHART network, specific functionality from the standard is a logical starting point. Our functional requirements deal with basic network configuration and transmission functionality, as can be seen in table 5.2.

FR01 Advertise correctly

In order for devices to initiate the join process of a network the devices need certain information, elaborated on in section 4.20. The information is provided in advertise packets transmitted by the Gateway at regular intervals. Our first goal is to properly build and transmit these packets, in order for devices to obtain the necessary information to initiate the joining sequence.

FR02 Join sequence

Once a device has received the advertise packets, a join request can be implemented. The join sequence is a specific process where communication goes between the new device, one of its neighbours and the Network Manager. FR02 deals with setting up the message passing functionality, where the goal is to have a device become fully integrated into the network.

ID	Functional requirement	Priority
FR01	Advertise correctly	Medium
FR02	Proper Join sequence	Medium
FR03	Proper transmission pattern	Medium
FR04	Functioning routing algorithm	High
FR05	Time synchronised network	High
FR06	Time Division Multiple Access	Low
FR07	Build MIC and CRC	Medium
FR08	Provide encryption of payload	High

Table 5.2: Functional Requirements

FR03 Proper transmission pattern

Asperheim et al. provided the Gateway with a hard-coded transmission functionality, where the packets were built on the fly and transmitted. After separating the Gateway into several layers as discussed in NFR01, we need to apply headers to the packet at the correct layers. On the receiving side, the headers will be stripped off the packets at the corresponding layers. This functionality is absolutely critical when operating with proprietary nodes, as communication will not be established if the transmission pattern is not according to the standard.

FR04 Functioning routing algorithm

Once a network grows beyond a few devices, more than one hop might be necessary before a packet reaches its destination. A packet can reach its destination in two ways; by flooding or by being routed. Since flooding is expensive and unreliable, we intend to implement a routing algorithm where a packet can be routed through the network successfully. We intend to implement a graph routing algorithm, before expanding it to support source routing as well.

FR05 Time synchronised network

Communication within a WHART network relies on time synchronisation in order to know when to transmit or listen for packets. Without proper time synchronisation, communication is based on chance. FR05 deals with implementing a timing functionality, where every device in the network have the same time and are able to successfully pass packets between them.

FR06 Time Division Multiple Access

TDMA greatly enhances the transmission quality in a network by providing simultaneous sending and opens for channel hopping as the clocks need to be synchronised. We intend to implement a transmission pattern where two devices can communicate by using several channels during one timeslot.

ID	Non-functional requirement	Priority
NFR01	Module-based applications	Medium
NFR02	Failure resistance	High
NFR03	Multiple Access Points	High
NFR04	Separate Access Points	High
NFR05	General security	Medium

Table 5.3: Non-functional requirements

FR07 Build MIC and CRC

MIC and CRC are two security mechanisms provided by the Data Link and Network Layers. They provide secure end-to-end communication and makes sure that no bit-errors occur within a packet. As a part of the proper packet transmission, we intend to implement these mechanisms to packets being transmitted within the network.

FR08 Provide encryption of payload

The data contained in a WHART packet is encrypted to ensure that no other devices than the recipient are able to read the content. An encryption algorithm based on AES-128 is discussed in the standard, and will be implemented to provide confidentiality to a packet.

5.2.3 Non-functional Requirements

Non-functional requirements are qualities a system should provide, and are often referred to as requirements a system must be able to provide a user with. We had certain ideas for what our system should be able to provide for users, listed in table 5.3.

NFR01 Module-based applications

The current system is tightly built, without any possibility to take out one part of the system and replace it with a new part without doing major changes. As WirelessHART is designed to be a layered protocol, we wanted to split the system into modules, where every layer provides one to many modules. Interaction between the layers would go through provided interfaces, limiting the amount of information passed between the layers.

NFR02 Failure resistance

We also want to make the system failure resistant, in case an Access Point or a node should fail. In that case the system will discover the error and reroute around these nodes.

NFR03 Multiple Access Points

NFR03 focuses on the ability to create and maintain several Access Points at once in a network. Asperheim et al. had one Access Point integrated in the Gateway, and we want to make it possible to have multiple Access Points operating at once. The possibility to run Access Points with and without time synchronisation abilities will also be addressed.

NFR04 Separate Access Points

Currently the Access Point is integrated into the Gateway, which means there is only one. We intend to separate the Access Point from the Gateway, making the Access Point able to run on its own platform without depending on the Gateway.

NFR05 General Security

Lastly, we want to provide some general security in the system. This would rely on several functional requirements, for instance FR06 and FR07.

5.3 Chapter Summary

The chapter has introduced the reader to the changes made in project goals during the timespan of the project. Reasons for changing the original project goals has been highlighted. The original goals as seen in table 5.1 has been modified to the functional requirements in table 5.2 and the non-functional requirements in table 5.3.

Chapter 6

Previous work

This chapter looks into the previous work done by Asperheim et al in their project. A brief description of the changes made on the Access Points, Gateway and Network Manager are provided. Their main field of work, the Atmel nodes, are not presented due to the reasons stated in chapter 5. Towards the end of the chapter, a test run of the code is explained.

6.1 Background

The work done on WirelessHART during the course of this project is based on the thesis "Design and Implementation of a Rudimentary WirelessHART Network"[3] by former students Kaja F. L. Skaar, Anders Asperheim and Rune V. Sjøen at University of Oslo in August 2012. Their work was a continuation of the master thesis "WirelessHART Gjennomgang og implementering" [7] by students Håvard Tegelsrud and Jørgen Frøysadal at University of Oslo in May 2010. Tegelsrud and Frøysadal implemented a complete Physical Layer on the AVR RZRAVEN nodes, including the send and receive functionality. Skaar, Asperheim and Sjøen used their work as a foundation and further expanded on their work. In their thesis they give a brief introduction to wireless networks and existing technologies, before delving into the WirelessHART standard in detail. In their implementation and conclusion sections, they elaborate on accomplishments obtained during the course of the project, described in the following sections.

6.2 Access Point

The next sections provides an overview of the implementation on the Access Points.

6.2.1 Adapting 15dot4-tools Sniffer

The Access Points provide an interface for sending packets on the network. The students started out with the Contiki OS but as they had to adapt the

project to work with WirelessHART, it was dropped in favour of 15dot4-tools. 15dot4-tools provides the user with the ability to sniff for traffic on the RavenUSB radio.

6.2.2 Receiving packets

The 15dot4-tools project is developed as a sniffer, and would therefore only be able to sniff and receive packets. As Asperheim et al. used the project both for transmission and receiving, they implemented a basic send module to support transmission of data packets. A switch were already existent in the program and by applying another case, the radio could be set to transmit from a command window in the same manner as setting the channel by **/ravenusb -d /dev/hidraw2 -c 20**.

6.2.3 Communication between AP and Gateway

The Access Points were programmed using C based HIDAPI, while the rest of the program were written in java. In order for the two parts to be able to communicate, a Java Native Library (JNI) were implemented which instantiates C methods and objects as java classes. By implementing the open source project called JavaHIDAPI, a non-blocking communication could be created between the Access Point and the Physical Layer of the Gateway.

6.3 Gateway

The following sections provide an overview of the Gateway implementation.

6.3.1 TX and RX queues

Modifications to the queue systems on the Gateway has been made. A global, shared RX queue is implemented as well as a separate TX queue for every neighbour.

6.3.2 Transmit and Receive Engine

The conclusion section states that a working transmit and receive engine is implemented in the program, where the packet is sent properly and received at another node. The packet is stripped of the related header at each layer, before being passed on to the layer above and responded with an ACK. When evaluating the code, the Gateway and Access Points had completed transmit engines but no receive engines were as of yet implemented. As mentioned earlier, the node implementation were not evaluated.

6.4 Network Manager

A Network Manager has been partly implemented with basic functionality. Structures for handling and storing link and neighbour relations are implemented, as well as superframes. Time Division Multiple Access has been started, but were not utilised due to the inaccuracy of the device clocks described earlier. For generating test results, values for creating a network were hard-coded in using the structures and methods already in place.

6.4.1 HART Command interface

The handler for processing a command and respond accordingly was implemented. As Asperheim et al. started at the lower layers, the handler has only been partly implemented due to limited time. The handler takes a command number as an argument, and fetches the corresponding command. So far only two commands has been implemented, *ReadLongTag* and *WriteLongTag*, which are not yet called by the program.

6.4.2 Graphical User Interface

A graphical user interface has been implemented to run on the Gateway to provide the user with a better overview of all the information provided by the program. The GUI provides four tabs and an additional command window:

Control - The control tab provides information about the network structure and communication between devices by displaying the network as a tree structure. The tree is regenerated every time the structure is expanded.

Graph - Provides a graph view by utilising the graph description language DOT to first create the connections between devices. Once all connections are established, the tool Graphviz is used to generate a PNG file from the graph which is loaded into the program.

Statistics - Holds statistics about the network, such as number of packets sent, cpu_usage, queues and more.

Live Incident Log - Records any major changes in behaviour on the Gateway components. Will log when a device is started, fail or close down.

Command Line - The command line provides a small interface for extracting information from the program. The commands *help*, *ping* and *numbers* are implemented. *Help* provides user guidance, *ping* contacts every node in the network and *numbers* displays the number of connected devices.

To track and gather the information neccessary, the program is configured to log every relevant information, which enables building the GUI on-the-fly.

6.5 Running the program

6.5.1 Initial tests of old code

When we ran the program handed over to us by the previous students for the first time, we met several challenges. The RavenUSB sticks were not marked, so actually finding the sniffer proved to be educated guessing. Configuring Wireshark did not prove as intuitive as we thought, as it had to be configured with a proper view and the correct channels. Once the Wireshark was configured and listening for traffic, the Network Manager was started and test packets were sent. We spotted spikes of traffic on several channels, some of it surely by other wireless devices in the room. However, certain channels proved to generate the same pattern of traffic over and over. We were certain that this traffic came from the Network Manager, as the generated traffic appeared at the same time as the it indicated sending packets. As the receive method on the Network Manager is not implemented, we had no way of testing if the nodes actually received the traffic and sent something back. As work on the Atmel nodes were discontinued, we decided not to spend time looking into what had been done here. We planned on reusing their work on the Gateway, Access Points and Network Manager, so they were the only parts of interest for us.

Soon after we ran the first test, the machines running Wireshark and the Network Manager broke down due to a hard drive failure. The Network Manager displayed a grub rescue error message, and the solution was to reinstall Ubuntu along with all the software needed.

6.5.2 Running code in new environment

After setting up the test environment again, we installed the java hidapi. The installation did not go as seamless as first intended, as several tools needed for the configuration were not present. In turn, they also needed configuring which were not documented in the packages or online. After modifying location and variables in some of the packages, we got eclipse to recognise the native libhidapi-jni-32 and libhidapi-hidraw. After getting past this obstacle, we felt certain that our Network Manager would now be able to communicate with the 15dot4-tools usb stick. However, when running the program we discovered that the usb stick is not recognised by the Network Manager during initilization. In the code we identify the devices connected to the machine when starting the program. The Network Manager fail to create an Access Point and to identify the 15dot4-tools usb stick, as can be seen in Appendix D.

After spending many hours on this problem and trying every solution we could think of, we decided to attempt running the manager on a windows machine instead. After installing the required software including the hidapi's and cygwin, we ran the manager and it instantly recognised the 15dot4 tools with the correct id. As some native linux methods were used for resource monitoring, they had to be rewritten in order to compile

the program. Going over to a windows machine from a linux machine required us to make some changes to the header of the packets being sent, as windows did not allow us to use the initial buffer in combination with the requested operation as can be seen in Appendix D. Not finding any documentation of this problem, we decided to contact the creator of the hidapi in hopes of getting an explanation of the problem. The problem turned out to be related to size of messages passed between a program and the usb interface, and is further elaborated on in section 7.4 on page 84.

6.6 Chapter Summary

The chapter has provided a brief overview of the work done on the network components Access Point, Gateway and Network Manager. The foundation our work is based on has been presented, and explained in section 6.5 where the execution of the code for the first time is explained.

Chapter 7

Development environment

The following chapter introduces the reader to the development environment of the project. Section 7.1 presents the different hardware used, while section 7.2 elaborates on the software. In section 7.3 and throughout the chapter, challenges met when establishing the environment are presented.

7.1 Hardware

The following sections present the hardware used during the project.

7.1.1 WiMon 100

The WiMon 100 is a wireless sensor made by ABB and used in different facilities and environments around the world. WiMon 100 consists of a vibration sensor, a temperature sensor, a battery and a WirelessHART radio. The device is designed to work in harsh environments such as the offshore sector, which goes hand in hand with the demands of our project partner Statoil. The device is mounted by a tapped hole at the bottom, making it possible to be installed at a wide range of locations. WiMon 100 are preconfigured with MAC addresses and response syntax, meaning we are not able to modify anything on the nodes as they work according to the standard. In table 7.1 on the following page we list relevant specifications of the WiMon 100 nodes.

7.1.2 AVR JTAGICE MK2

AVR JTAGICE MK2 is the debug tool developed by Atmel for on-chip compiling and debugging. JTAGICE is connected to a computer through a usb interface and to the devices by a debugWIRE interface. JTAGICE mk2 is supported by AVR Studio 6 (see section 7.2.4), enabling us to upload our program to the RavenUSB and modifying the code on the raven boards through the IDE. However, as we acquired new nodes, we only used JTAGICE for debugging code at the RavenUSB nodes.

Data processing	
Velocity, range	10Hz - 1KHz
Envelope, range filter	500Hz to 10KHz
A/D conversion	16 bit
Measuring schedule	Remotely configurable
Wireless Communication	
Network standard	WirelessHART (HART 7.2)
Radio standard	IEEE 802.15.4
Frequency	2.4 GHz, licence free ISM band
Output power (peak)	<10 mW
Range (nominal)	>50m @ line-of-sight
Physical	
Weight	0.2 kg
Height	10 cm
Case material	Stainless steel/Thermoplastic
Mounting	1/4 28 UNF tapped hole
Dimensions	100 x 36 mm
Misc	
Battery lifetime	<5 years with waveform upload interval <1/day and vibration rms and temperature values upload interval <1/hour
Recommended number of associated sensors	100
Communication protocol	HART over RS-485, HART over UDP MODBUS RTU/TCP

Table 7.1: WiMon 100 specifications

Figure 7.1: Wimon 100

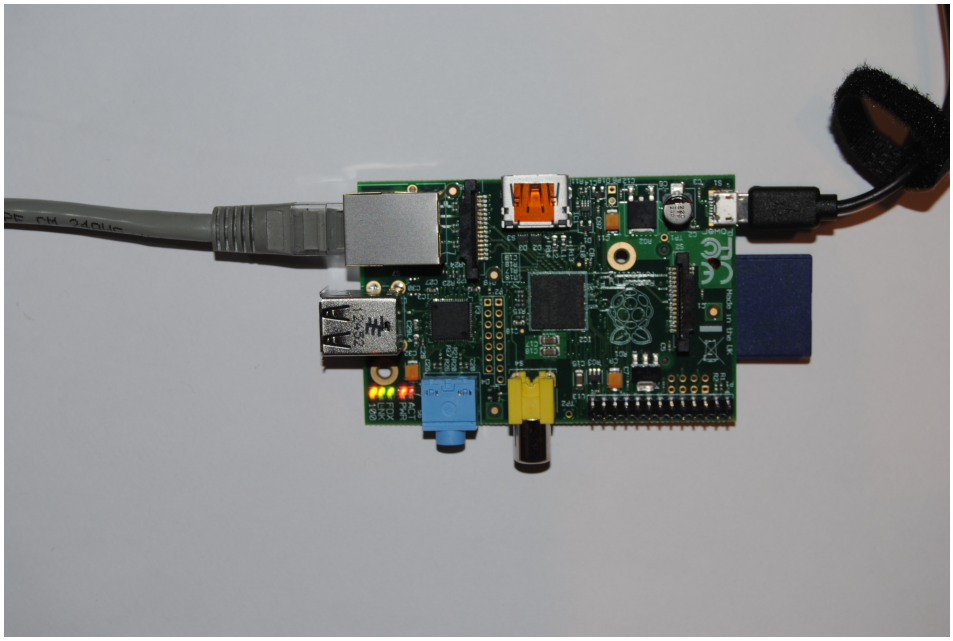

Figure 7.2: Raspberry Pi

7.1.3 AVR RZ Raven P/N ATAVRRZRAVEN

The kit includes two AVR Raven boards and a usb stick for connecting to a computer. The Raven card has a 2,4 GHz receiver, a picopower ACR program processor and LCD screen. Atmel's radios are developed for the Zigbee standard, but are capable of transmitting WirelessHART traffic as both are based on 802.15.4 standard. The AVR Raven radios were used by the previous students, but proved insufficient for meeting the clock demands by the WirelessHART standard. We have only used these radios for testing traffic monitoring on Wireshark (se section 7.2.1). The RavenUSB sticks work very well with the sniffer software however, and will be used as they are.

7.1.4 Rasberry Pi

Raspberry pi is a single-board computer at the size of a credit card developed by the Raspberry Pi foundation. The computer is manufactured through deals with the companies Element 14/Premier Farnell and RS Components, and only sold online. It does not contain any extra hardware, meaning that no hard drive is supplied. Booting and storage is performed on a SD card. Relevant specifications are listed in table 7.2 and the device can be seen on figure 7.2. The computer is used by enthusiasts to perform different minor tasks, and proved to be a helpful tool for our emulations. We built a network using two raspberry Pi's connected to a hub over ethernet.

Specification	
CPU	700 MHz ARM11,
GPU	Broadcom VideoCore 4, OpenGL ES 2.0, MPEG-2 and VC-1, AVC high-profile decoder and encoder
Memory (SDRAM)	512 MB (shared with GPU)
USB 2.0 ports	2
Video outputs	Composite PRCA, HDMI, raw LCD panels via DSI
Audio outputs	3.5 mm jack, HDMI, l2S audio
Onboard storage	SD/MMC/SDIO card slot
Onboard network	10/100 Ethernet usb adapter
Power ratings	700 mA (3.5 W)
Power source	5 volt via MicroUSB or GPIO header
Size	85.60 mm x 53.98 mm
Weight	45 g
Operating systems	Debian GNU/Linux, Raspbian OS, Fedora, Arch Linux ARM, RISC OS, FreeBSD, Plan 9

Table 7.2: Raspberry pi specifications

7.2 Software

The following sections present the software used for code development.

7.2.1 Wireshark

Wireshark is a network protocol analyser that started out as a small project in 1998. Over the years it has grown to become the de facto standard in many industries and educational institutions due to the contributions of network experts [19]. As it is free of charge and holds a high quality, it is often preferred over commercial sniffing tools. Wireshark supports a large number of protocols, but not WirelessHART specifically. As 802.15.4 and Zigbee traffic is supported, it is possible to transmit packets with Zigbee headers and strip off those headers before processing the packet. Wireshark captioning *Command* and *Ack* packets can be seen in figure 7.3.

7.2.2 AVR2025

The AVR2025 software package provides a code library for development on the Atmel MAC architecture supported by all Atmel IEEE 802.15.4 chips [4]. The code library allows flexible firmware configuration, supports different microcontrollers and allows easy platform porting. As it also supports Atmel AVR Studio (listed in table 7.3), we can easily develop code based on this library in AVR Studio 6, and then port the code to the Atmel usb sticks used as Access Points.

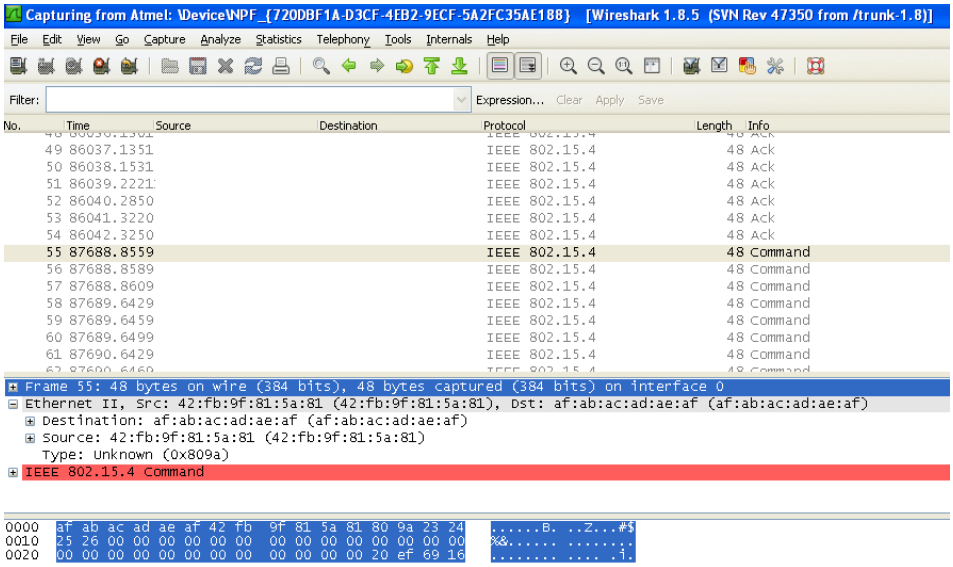

Figure 7.3: Wireshark

7.2.3 The 15dot4-tools Project

"The 15dot4-tools project is a set of valuable 802.15.4 tools. They are open-source tools, allowing a community effort to develop them." [1] This project is written in C and Assembler. It consists of a set of open-source tools for sniffing, sending and analysing packets sent over the 802.15.4 standard. The sniffer is available for the RavenUSB-stick (see section 7.1.3) we are using, but the radio tools are only available for the Dresden Sam7s stick. This created a challenge for Asperheim et al. on this subject and will be discussed further in section 7.3.2.

7.2.4 Additional software

We have used a few programs for small parts of our project. Due to their limited role, we have listed them in table 7.3 along with a brief description.

7.3 Encountered challenges

The following sections provide short explanations of encountered challenges.

7.3.1 Evaluating the state of the project

Over the course of the project we encountered several issues. The first challenge was that we had no idea how the network was supposed to be put together. We had gotten an idea by reading the master thesis of the previous students and through conversations with them. When we began the project, the entire network was disassembled and nothing was

Software	Version	Use
Eclipse Juno	4.2	Programming Java and C in an IDE providing easy handling of larger project files. Enables use of plugins for synchronising work through github and SVN.
JUnit testing	4.11	Tests used within our program to check if correct values are assigned and the correct program syntax is followed.
Subversion	1.7.9	Software to provide revision control and software versioning. Used until Github proved to be a better option.
Github	1.0.41.2	Version control software. Provides a clean interface giving the user an easy overview of revisions. We favoured Github due to the amount of control that is provided to the user.
Apache Maven	3.0.5	Builds java projects dynamically by downloading java libraries and maven plugins defined in a build.xml file attached to the project. Reduced the risk of errors when we moved the code between machines.
Atmel Studio 6	6.1 beta	Integrated development platform for developing and debugging Atmel AVR microcontroller-based applications[5]. Were used to compile via the AVR-GCC compiler and upload the compiled code to the Raven-USB device.

Table 7.3: Additional software used

marked. After reading online guides we got the environment properly set up, although temporarily which is detailed in section 6.5 on page 72. We were handed six unmarked RavenUSB sticks, two of them running as sniffers. After some testing, we found one that were reacting differently to traffic sent from the raven boards and determined that it was a sniffer. The rest of the nodes we recompiled the code for, as we were unsure what they actually contained.

7.3.2 Problems compiling 15dot4-tools

15dot4-tools are used on our RavenUSB sticks to be able to send and receive data from the sensor network discussed in section 7.2.3. In the master thesis "Design and Implementation of a Rudimentary WirelessHART Network" [3] Asperheim et al. modified this project to be able to send 802.15.4 packets over the RavenUSB-stick using the 15dot4-Sniffer-tool as a base. Configuring RavenUSB-stick with 15dot4-tools were not documented anywhere, making it a challenge for us to identify the problem. The issue arose due to a new compiler released since the last time the code was used. To compile this project we needed to use AVR Studio 6 (see table 7.3) together with the AVR JTAGICE MK2. The code was last compiled with the older AVR Studio 5 and with the new AVR Studio came a new AVR-GCC compiler. The new compiler does no longer accept non-constant variables in the read-only section of the program memory. Once we discovered the problem, it was easy to solve the issue by going through the project and changing the needed variables to constants. After rewriting parts of the code, the program worked properly again.

7.3.3 Compiling 15dot4-tools

To compile a legacy AVR Studio 5 project we needed to create a new project from the Makefile with an extension called "Create Project From Makefile (Beta)", and configure this project to use the external Makefile we used to make the project. We also needed to choose the correct device we would like to use, in our case this is the AT90USB1287[1].

7.3.4 Configuring Wireshark

The following sections provide an overview of the challenges met when configuring Wireshark for sniffing traffic.

Ubuntu

The former students working on the project had been running Wireshark on a linux redhat machine in combination with a lua dissector[2] for analysing the packets transmitted. The development environment went down shortly after taking over the project, which is further described

[1] The RavenUSB device

[2] a small script for investigating packets

in section 6.5. When reconfiguring Wireshark, we discovered that synchronising all the software and hardware was not intuitive. We first attempted to run the program on an ubuntu machine. Wireshark were set up properly, but could not find the usb stick when selecting which interface to monitor. After reinstalling the hidapi and hidraw api drivers and their dependencies, we were able to find the ravenUSB when moving through the directories to **/sys/class/hidraw/hidraw2/device/uevent**. Using the command **/ravenusb -d /dev/hidraw2 -c 20** we were allowed to set the device to listen at channel 20. Wireshark would not recognise the RavenUSB as an interface however. When running a test program to check which hid devices were found, the RavenUSB were not one of them as can be seen on figure 7.4. After trying every option that came to mind, we decided to abandon Ubuntu and try a different approach.

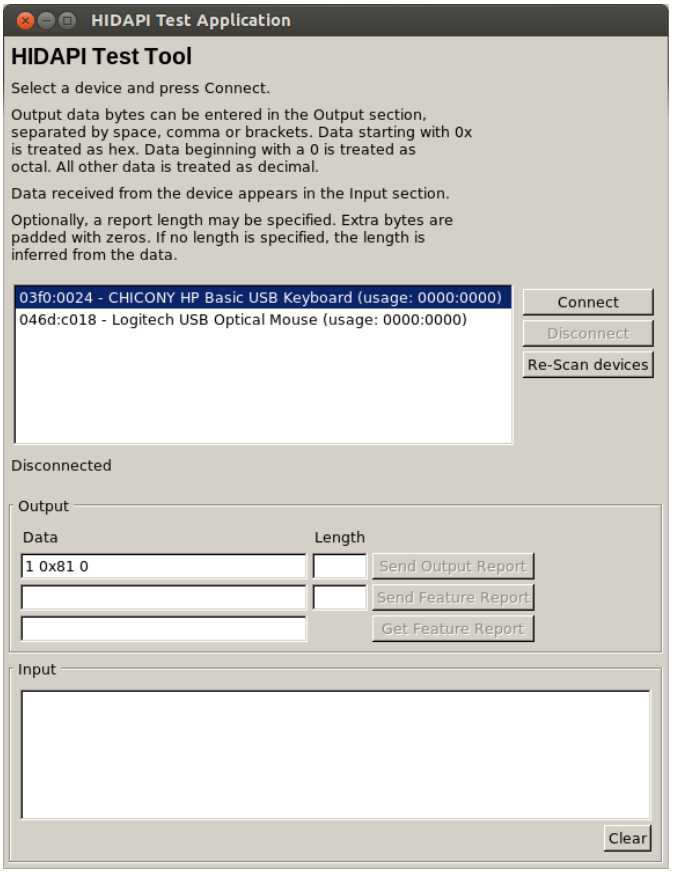

Figure 7.4: HidAPI test application

Windows 7

After setting up Windows 7 on our test machine, we installed all the required drivers without any problems. Wireshark was running, and we had an additional java program (atUsbHidGui) for setting the correct channel for the RavenUSB, acquired from the website of the maker of

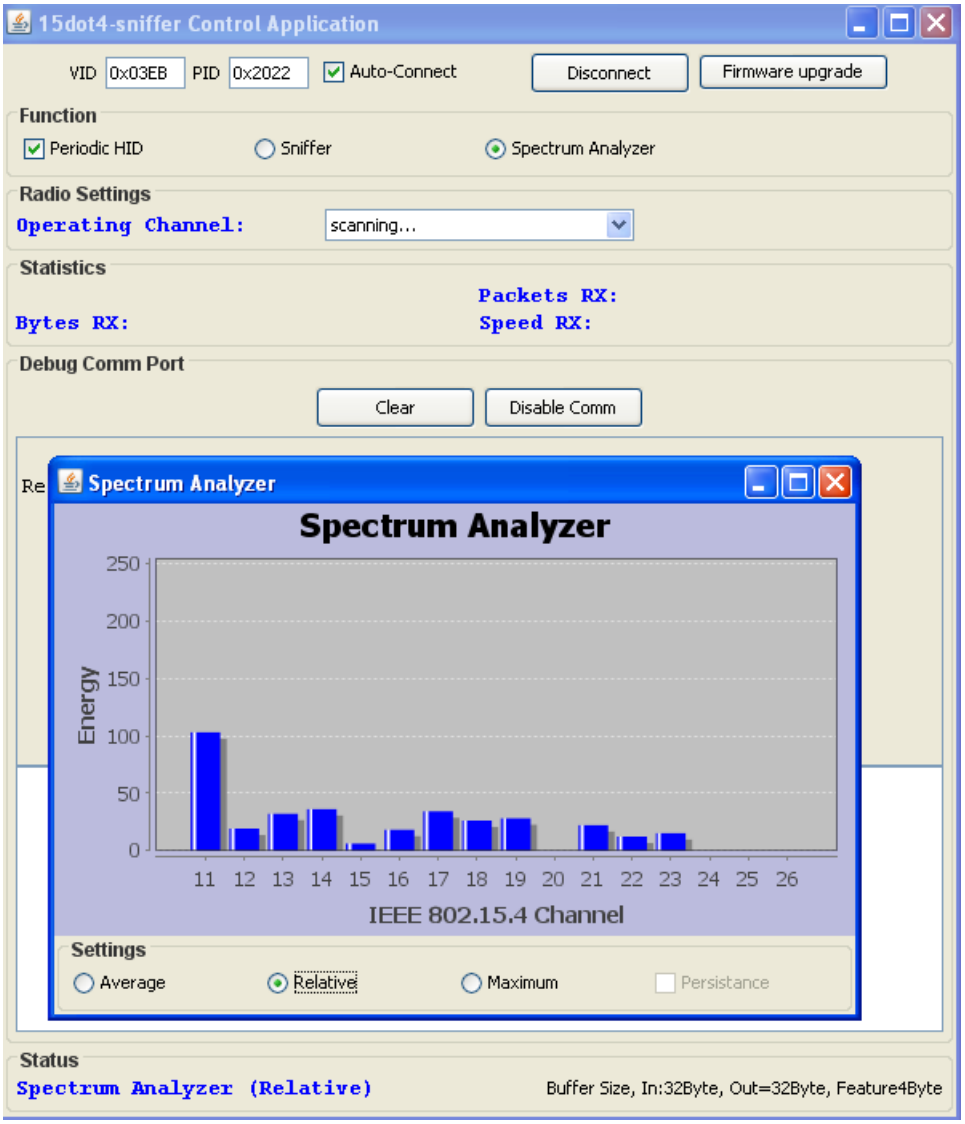

Figure 7.5: atHidUsbGui for setting channels

15dot4-tools [1].

atUsbHidGui seen in figure 7.5 displayed traffic on different channels, making us confident it was working as intended. However, the program crashed after a short while caused by an error which made the program unable to run again. We encountered the same problem as on Ubuntu; Wireshark could not recognise the RavenUSB as a network device. We sent a mail directly to the maker of 15dot4-tools, Colin O'Flynn, to ask what might cause the problem and which operating system would be the optimal solution. The reply was that it might or might not work on Windows 7, as the project was discontinued in 2010.

Windows XP

After spending several hours on this problem, we determined that we would go back to an operating system that were guaranteed to run the software without any problems: Windows XP. After going through the setup for a third time, we followed a guide on Colin's website [1] for windows configuration of the usb stick. Where Windows 7 did not recognise the RavenUSB as a new device, Windows XP did immediately. After installing the *CompositeAtmelRNDIS.inf* driver, RavenUSB appeared as a new network connection. Disabling every item under its properties and then running Wireshark proved to be working perfectly. As the Raven commands were troublesome to install on Windows 7, we used Colin's *AtUsbHidGui* for setting the channel with great success. Shortly after starting the Network Manager, Wireshark started to display traffic.

7.4 From Wimon100 to Raspberry pi

During the project we had to change our way of transmitting packets in order to gain test results as described in chapter 5 on page 63. The Wimon100 nodes (section 7.1.1) acquired from ABB were used for a short time only before the program were restructured and rewritten to strictly follow the protocol specifications. Asperheim et. al sent small test DLPDU's where the headers were hard-coded and did not contain any payload. Once we had written the layers to apply correct headers to outgoing packets, and started transmitting traffic, we encountered an error from the java HidApi[3] seen in listing 7.1.

```
1 java.io.IOException: The supplied user buffer is not
    ↪ valid for the requested operation.
2 at com.codeminders.hidapi.HIDDevice.write(Native
    ↪ Method)
3 at HIDAPITest.readDevice(HIDAPITest.java:57)
4 at HIDAPITest.main(HIDAPITest.java:29)
```

Listing 7.1: hidapi problem

Java HidApi only supports transmission of 64 bit packets where 32 are already used. After contacting the developer of HidApi [15], Alan Ott, he proposed that we could write a Libusb jni/jna wrapper[4]. We realised that we would not be able to communicate with the Wimon100 nodes before they received a packet identical to the standard specification and responded to it. In addition, we would not be able to send the proper packet with the current setup. Writing the wrapper in itself is a whole new project, and acquiring the new usb sticks would take too much time. We decided to abandon the Wimon100 nodes for now, and move over to an approach leaning towards simulation. We still wanted to send packets over

[3] Java framework enabling communication with the usb devices

[4] Programming framework enabling java code to call and be called by native applications and libraries written in other programming languages.

a network, and looked around for solutions or set-ups that could work. Raspberry pi, introduced in section 7.1.4, provided us with the possibility to run a simple program on a device acting as a node. By running several nodes on the Raspberry pi listening on different ports, the simulation would resemble a WirelessHART device to a certain extent. Every aspect of energy restriction and limited resources were removed, but we could test and evaluate the capabilities of packet construction, communication and routing.

7.5 Chapter summary

This chapter has presented the development environment of the project to the user. The different hardware devices and software tools have been introduced, along with encountered challenges related to establishing the development environment.

Chapter 8

Implementation

This chapter presents the implementation part of the project. The chapter is split into sections according to the network components, with cross-device functionality towards the end. Section 8.1 details how the overall structure of the project has changed and how the communication flows between the layers. Section 4.2.3 on page 26 details the implementation of the Gateway and how it has been separated into layers. Section 8.3 presents the implementation of the Device Network Management module containing information shared by devices in the network. Sections 8.4 to 8.6 discuss the implementation of the Network Manager, Security Manager and Access Points. Section 8.7 on page 105 details the implemented security for a transmission, while section 8.8 presents the construction of the DLPDU.

8.1 Program structure

The program structure has changed to an approach closer to the standard over the course of the project. Asperheim et al. worked on every aspect of the WirelessHART network. Due to limited time they integrated the Network Manager, the Security Manager and the Access Points into the Gateway. The different parts were running on the same machine, split into separate processes. Our first task were to separate the Access Point from the Gateway as seen in figure 4.3 and discussed in section 8.6.1. The Access Points are only communicating with the Physical Layer at the Gateway requiring data traffic to go the correct route from layer to layer before being transmitted to the Access Points.
As seen in figure 8.1, the old project design held one Access Point and used the Raven USB running 15dot4tools to communicate with the Atmel AVR devices in the network. In the new design shown in figure 8.2, the Gateway, Network Manager and Security Manager are still running on the same machine as different processes. Interaction with the Security Manager goes through the Network Manager now however, as the Gateway shall not be aware of the Security Manager according to the standard.
The Access Point has been separated from the Gateway, and the network are now able to run several Access Points simultaneously. Section 8.6 further elaborates on the changes made on the Access Points, but major

changes include separate queues between the Gateway and every Access Point. The Access Points can now act as a time synchronising source, but depending on the size of the network, not everyone are required to do so.

8.1.1 Communication between the layers

The WirelessHART layered approach is based on the OSI model. In both standards, interaction between the layers are performed through interfaces, discussed in section 4.3 on page 27. As mentioned in the previous section, we restructured the program into a layered approach. We wanted all communication between the layers to to go through interfaces, in order to separate the tasks of every layer clearly. The standard specifies a set of interface calls for every layer listed in section 4.3. Every interface call has not been implemented, due to time restraint and need-basis. We implemented the calling syntax between the layers however. If an upper layer requires a service for a lower layer, a *request* is sent and is responded to with a *confirm* from the lower layer. If the lower layer needs to notify an upper layer of an event or a received packet, an *indicate* message is passed up the stack.

8.1.2 The Common Packet Structure

Besides separating the major parts of the program, every layer has been separated into clearly defined modules. We determined that clearly separating the functionality into smaller modules, would provide a better overview of the program while at the same time making it easier for the reader to navigate in the code. The modules for every device are described in their corresponding sections later in the chapter. Below we describe the common package which contains help methods utilised by all the other packages.

Common - Contains help methods for value conversion when data traverse the layers and are transmitted between devices. Also contains a nickname generator used by the Network Manager when a new device is joining the network.

Common.Stack - Contains a set of interfaces for the managers and every layer in the Gateway. Every module contains the necessary values and methods the inheriting class needs to implement.

Common.Connection - Contains information necessary for establishing the test network and creating links between neighbouring nodes.

8.2 Gateway

The Gateway has undergone several changes since the previous thesis. The Gateway used to have integrated Access Points. Communication with the Security and Network Managers did not follow a specific pattern, but were

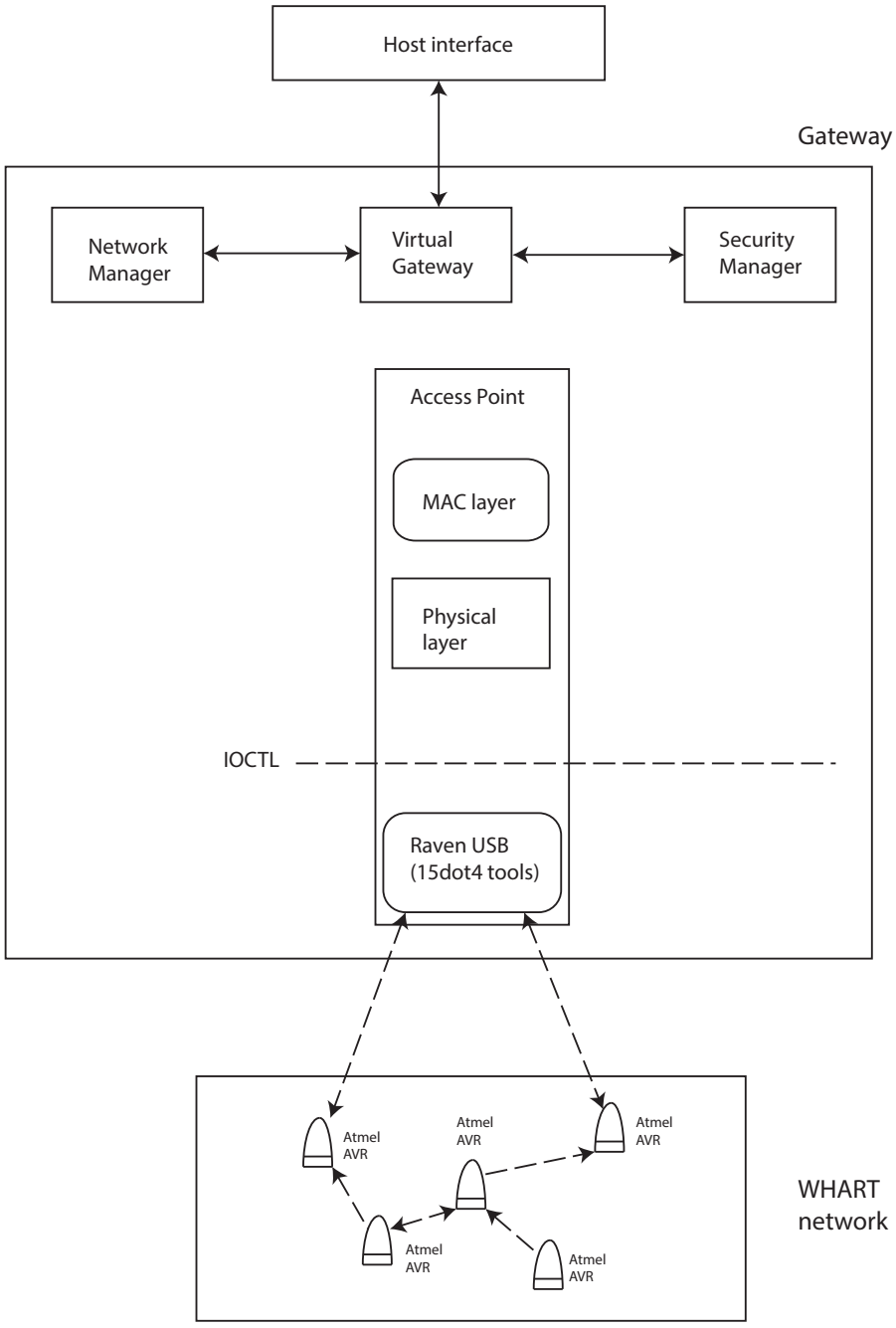

Figure 8.1: Old design

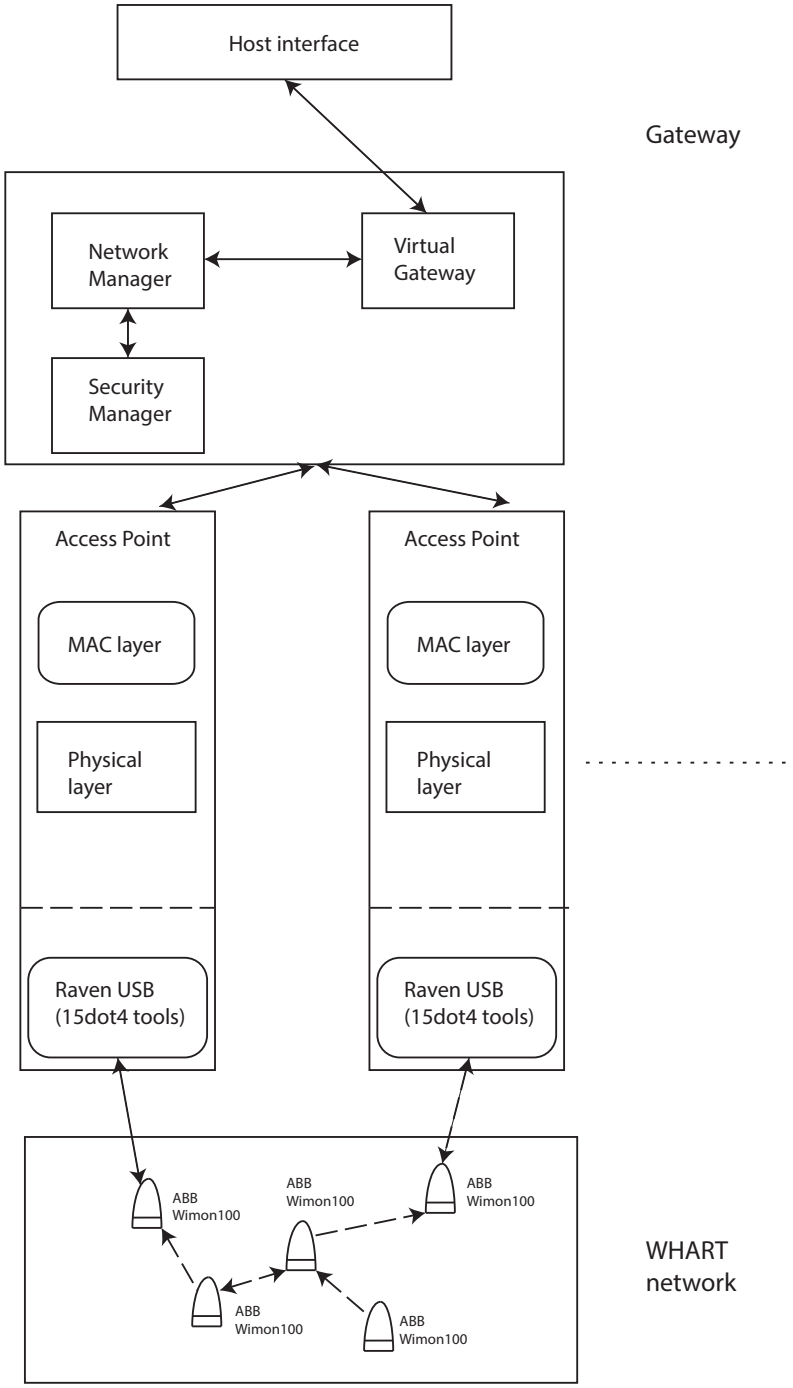

Figure 8.2: New design

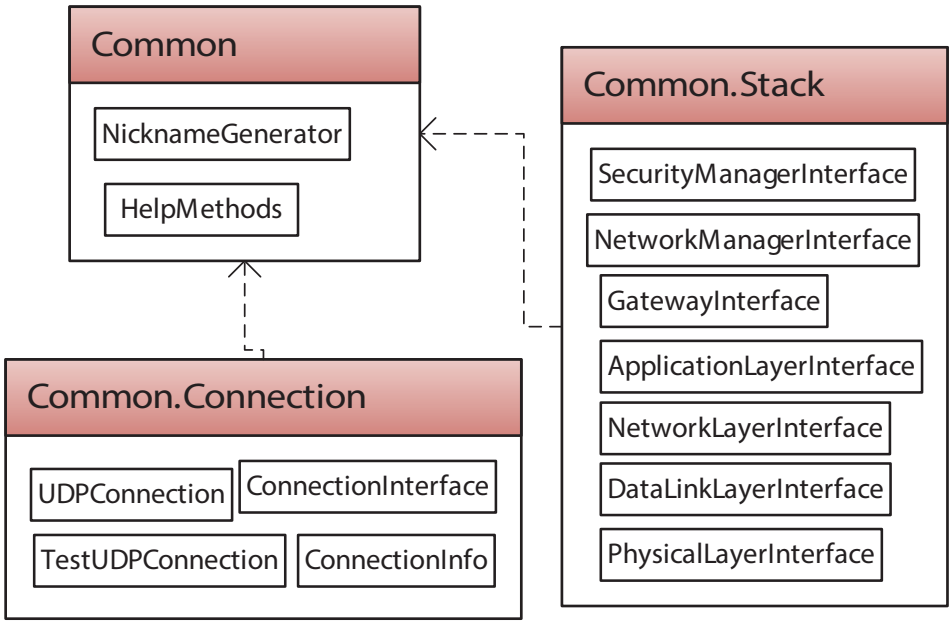

Figure 8.3: Common Packet structure

based on the needs of the Gateway. The Gateway contained the queue systems for the Access Points and generally performed every task in the network by itself. We decided to split the Gateway into separate layers and delegate tasks according to the protocol specification. The queue systems were moved out to the Access Points as specified in section 8.6. The Security Manager connection were moved to the Network Manager, and interaction with the NM were included in the Data Link and Network Layers as can be seen in figure 4.3.

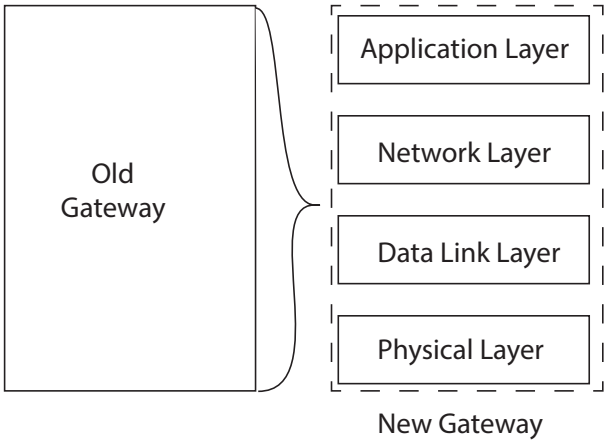

Figure 8.4: Gateway changes

8.2.1 Gateway Packet Structure

The following sections provide an overview of the packet structures of the Gateway shown in figure 8.5 on the next page. The solid arrows depict the calling sequence.

ApplicationLayer - Implemented according to the Application Layer in the standard. Contains modules for building and processing commands.

ApplicationLayer.Startprogram - The Startprogram contains the necessary information for starting a device in the network. The module determines what type of device it is running on, and starts the device accordingly.

ApplicationLayer.Commands - A help package for the Application Layer. Contains currently implemented commands, separated to make the system easy to maintain and scale.

ApplicationLayer.commands.Wirelesslesshartcommands - Contains commands specific to the wirelessHART standard, and not available on HART devices.

NetworkLayer - Implementation of the Network Layer. Attaches and detaches the NPDU header.

Datalinklayer - Package containing the functionality of the Data Link Layer. Contains modules for handling PDU's, schedule links and keep track of timouts. The DLL also builds packets with the appropriate values as well as providing the correct sequence for handling incoming packet types.

PhysicalLayer - Represents the Physical Layer. Currently contains the module needed for simulating channelhopping over ethernet.

8.2.2 Timer

Timing is essential for a WirelessHART network to be able to function and communicate correctly. The timing module is located at the Data Link Layer, and is used in combination with the state machine to determine the next event. The different timer types can be found in HCF Enumeration table 43[1], which we do not have access to. As a consequence, we have implemented the timer types mentioned in the standard along with specified values as can be seen in table 8.1. -1 denotes a currently invalid value. *Command 795 writeTimerInterval*[2] is used for setting the time for each timer type, and can be used to change the interval of each timer. It is frequently used during start-up of the network, where the advertisement interval is short and is changed to a longer interval after the initial phase.

[1] Tables available for purchase from Hart Communication Foundation

[2] For full information refer to appendix B on page 143

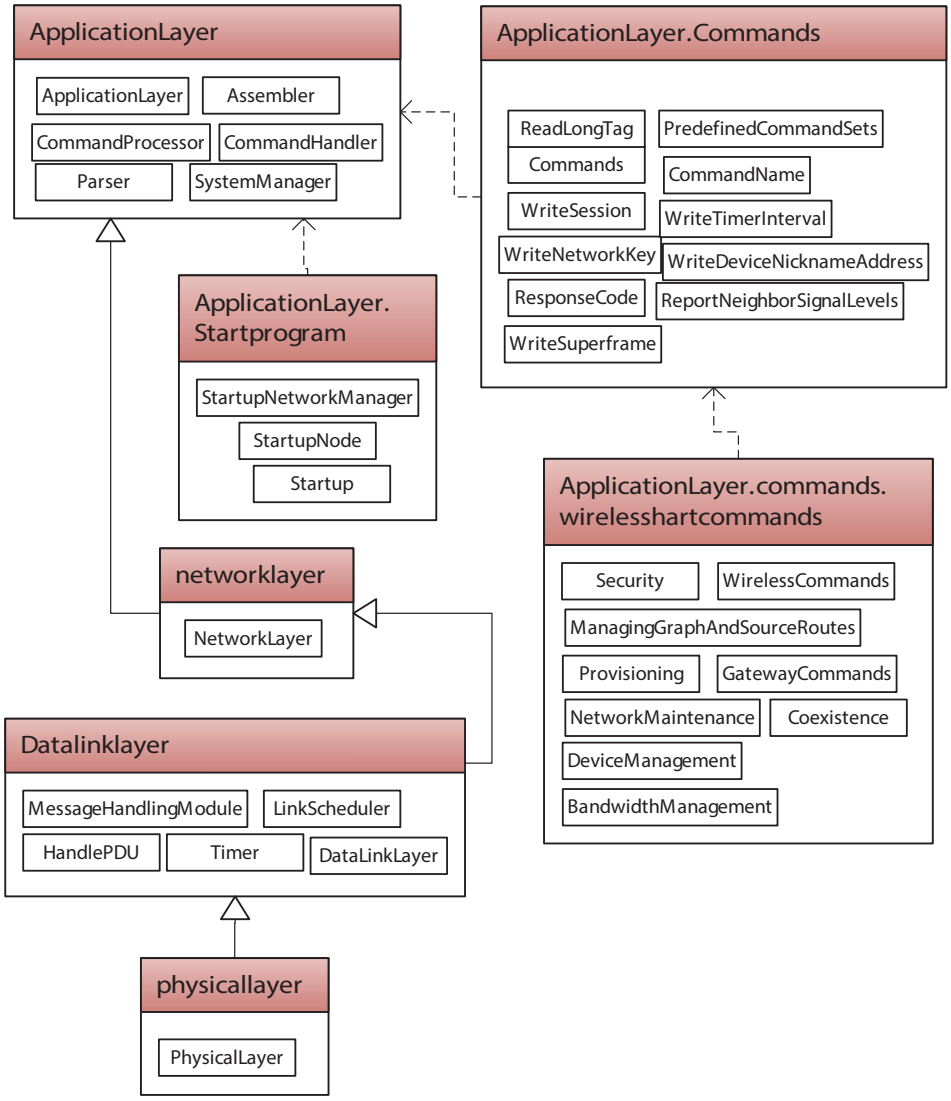

Figure 8.5: The layer structure

Timer type	Default time
ACTIVE_SEARCH_SHED_TIME	4.000.000 ms
ADVERTISEMENT_INTERVAL_TIME	-1 ms
ADVERTISEMENT_WAIT_TIMEOUT	30.000 ms
BROADCAST_REPLY_TIME	60.000 ms
CHANNEL_SEARCH_TIME	400 ms
DISCOVERY_INTERVAL_TIMER	-1 ms
HEALTH_REPORT_TIMER	900.000 ms
JOIN_RESPONSE_TIMEOUT	30.000 ms
KEEP_ALIVE_INTERVAL	30.000 ms
PASSIVE_CYCLE_TIME	600.000 ms
PASSIVE_WAKE_TIMER	6.500 ms
PATH_FAILURE_INTERVAL	-1 ms
MAX_PACKET_AGE	300.000 ms
MAX_REPLY_TIME	30.000 ms

Table 8.1: timer types

The timer module handles all timing issues in the system and is used to determine the next occurring event. The module determines the timing type handed to it by the state machine, and acts accordingly. For every case it is important to verify that the time is lower or equal to the ASN (see section 4.8.1 on page 48), to avoid processing packets that are delayed and have missed their timeslot. Once the timing type has been determined and the time checked, the case is executed before the ASN is incremented. In the test advertisement case, the control is handed over to the *message handling module* which creates a new advertisement packet. The advertisement is transmitted, before the ASN is incremented. In the event of a packet failing the timer check, the packet is considered too old and is removed from the queue. The same goes for packets not ACK'ed in time, as they will have to be discarded and resent.

8.3 Device Network Management

Every WirelessHART network has a container for storing information shared between devices, as can be seen on figure 8.6.

DeviceNetworkmanagement - Contains data structures necessary to store and maintain the network topology.

DeviceNetworkManagement.Queue - A large amount of queues has been implemented. Every queue type is implemented as a separate module, to distinguish between the packet types queued.

DeviceNetworkmanagement.Pdu - The Pdu package contains the different packet types described in section 4.5.6 on page 38. Every module contains the necessary information for generating the associated packet type. A module for handling the payload is also present.

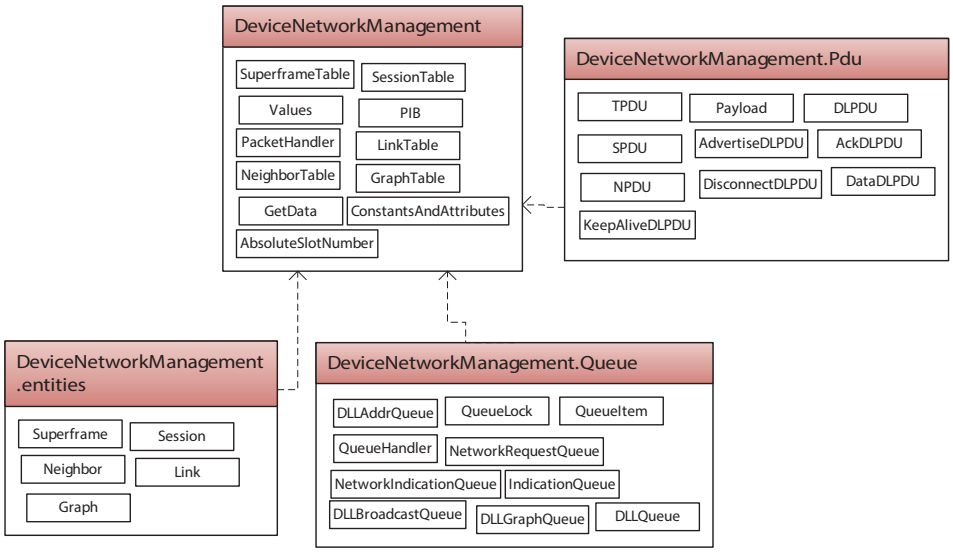

Figure 8.6: Device Network Management Packet Structure

DeviceNetworkmanagement.Entities - Defines the different structures needed for keeping a network topology. The containers are specified in *DeviceNetworkManagement*.

8.4 Network Manager

The Network Manager's connections have been modified, and can be seen in figure 8.2. In the following sections we provide an overview of the changes made on the NM.

8.4.1 Network Manager Packet Structure

NetworkManager - Contains a single module for initializing the Network Manager.

NetworkManager.Nodes - The Network Manager stores information about every network device in a nodecontainer. Also contains a module for routing packets to devices.

NetworkManager.Graphs - Stores information about the network graph.

8.4.2 Advertisement

To form a network, the Gateway transmits advertisement packets at specified intervals. The timer module contains an enumerate *ADVERTISEMENT_INTERVAL_TIMER* set to three seconds. Every three seconds the timer module will timeout the enumerate and send a new advertisement on the medium. Once a node receives an advertise, we calculate the

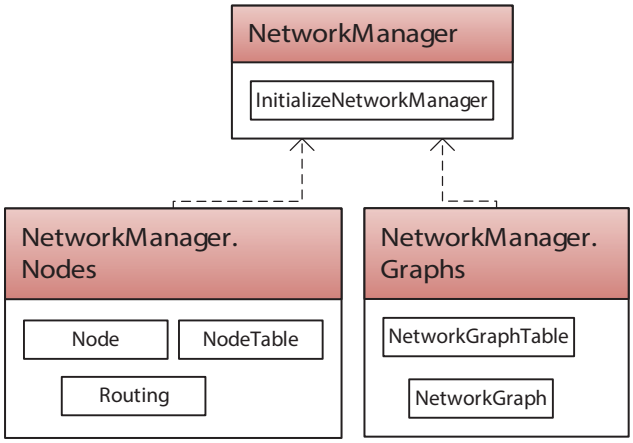

Figure 8.7: Network Manager Packet Structure

ASN at which it was sent and check if the Network Layer has started the join process. If that is the case, we initiate the timer *ADVERTISEMENT_WAIT_TIMEOUT* to 30 seconds. During this time, we wait for additional advertisement packets to arrive. If no further advertisement packets arrive, the device returns to active search again. A node will stay in active search for 4000 seconds, before going over to passive search according to the standard. In our implementation, we have omitted the passive search as it was not considered important at this stage.

8.4.3 Join process

All commands mentioned in this section is thoroughly explained in Appendix B on page 143. When a node is presented with the Network ID and the Join Key for a network, it starts listening for Advertise packets. When the first Advertise packet is received, the joining node synchronise its clock to the clock of the advertiser before activating *ADVERTISE_WAIT_TIMEOUT* in the Timer class. During this timer the node will listen for more advertisements trying to build the neighbour table and calculate the average RSL[3] of all its neighbours. Once it times out the join process is initiated. The join process is initiated by sending a join request on one of its neighbour's join links that are stated in the Advertise packet. The request contains the commands *20: Read Long Tag* and *787: Report Neighbor Signal Levels.* The Network Manager receives the join request. With the information received, it decides on what node should be the node's parent for more traffic and generates a nickname for the node. This information is then used to create the join reply, containing the three commands *Write Network Key, Write Device Nickname Address and Write Session.* The join reply is then sent to the neighbour the Network Manager has decided as the joining nodes parent, with the proxy bit set in the Network Layer PDU. In our implementation the Network Manager choose the neighbour by getting the first neighbour in the list.

[3]RSL - Received Signal Level

It could be implemented in several ways like closest neighbour by jumps, or the node with the least traffic. Once the node receives the join reply, it is in a quarantined state, meaning it can communicate only with the Network Manager. To get to the operational state the node need to receive superframes, graphs and sessions from the Network Manager.

8.4.4 Superframe calculation

In order to provide reliable communication, WirelessHART schedules the medium access through superframes, discussed in section 4.5.1 on page 36. A superframe holds a number of slots, where each one can be assigned to a link. A link has a transmitter and one to many receivers, depending on the type of communication scheduled for the slot. The protocol does not specify the length of a superframe, only that it can be of variable length. It is the task of the Network Manager to assign which superframes are active, meaning a device can hold inactive superframes for later use.

In order to keep track of the superframes, a *superframeTable* was created with a list of all superframes and a maximum number of 20 superframes in the system. This restriction is mentioned in the standard, but with no reason as to why. We guess this is because of the limited resources on the nodes.

To get a superframe to a device in the network *command 965; write superframe* is called. The command transfers information like the superframe ID, slots in superframe and an execution time for the command. Links are transferred with a separate *command 967; write link*. These commands are further explained in appendix B on page 143.

In our implementation the superframe calculation is very basic. All nodes in range of each other get one data link to communicate and there is one join link for each device as well as a discovery link. Broadcast links are not implemented yet. For more details on the link types see section 4.5.2 on page 36.

8.4.5 Routing

In WirelessHART there are three types of routing used; graph and source routing discussed in section 2.1.5, and superframe routing which is not implemented. The type of algorithm used is determined by whom the sender is. If a node is sending information to the Network Manager as the recipient, graph routing is utilised. If the recipient is another node, or the Network Manager sends a packet to a node, source routing is used. We decided to implement a graph routing algorithm first in order to find all available paths. Once all the available paths are discovered, using sub-trees of these we can build the source routing as well.

Implementing graph routing

When researching routing algorithms for mesh networks, we came upon an article about a graph routing algorithm suitable for the requirements

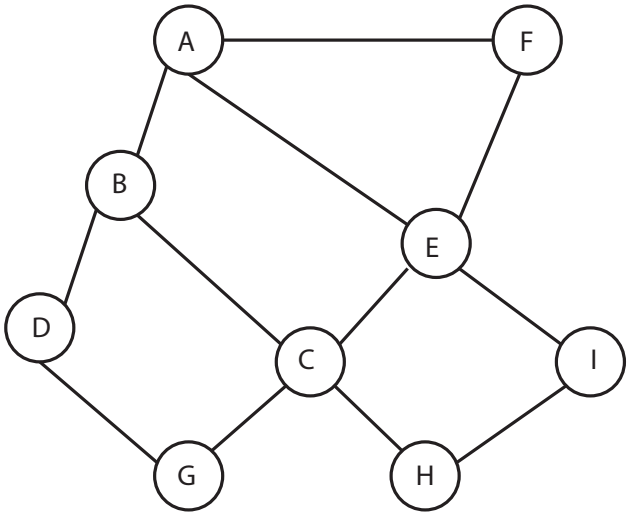

Figure 8.8: Example topology

of WirelessHART [13]. The protocol named Exhenced Least-Hop First Routing (ELHFR) determines the network topology as a connected graph and then generates the shortest paths. The shortest paths are only determined for leaf nodes, as a sub-graph of any leaf would be the fastest route to a node part of the route. An example can be viewed in figure 8.8. We have a partial mesh network, with several routes to each node. The routing algorithm discovers the shortest path to every leaf node, currently nodes *G*, *H*, *I* and *F*. A breadth-first tree[4] is constructed as can be seen in figure 8.9. Complete routes to every node are now generated. As we want to use the algorithm for building our source routes as well, we decided to generate all routes for leaf nodes first, but also store every sub-graph possible.

New device joining

The suggested routing protocol provides one case for the network creation and one for when a new device is joining. As our network is small and will be started from scratch, we decided to merge the two cases with the possibility to expand it with an additional update function later. Part of the reasoning behind this solution, is that every new joining device will have two parents. The parents' neighbour tables will have to be updated accordingly, as well as the new node. Doing a complete recalculation of the entire network while it is of a small scale causes low overhead.

As seen in listing 8.1, we first check if we deal with a leaf node. As the joining device is always a leaf node, this will always be the case. For every neighbour, we create a graph table containing the route to the Gateway. If several routes are available for a neighbours, the neighbour is stored with each path. For every one-hop neighbour, we get its neighbours and

[4]A tree structure where root is searched first. Afterwards all its children are searched, followed by their children etc.

```
updateNewDevice(deviceNode newNode, boolean isLeaf,
GraphTable currTable){
  if(isLeaf){
    for(Neighbor n: newNode.returnNeighbors())
      ArrayList<Neighbor> refNeighbors = new<Neighbor
        ↪ >
      ArrayList<Neighbor>();
      GraphTable newRoute = new GraphTable(
        ↪ genGraphName(),
       0, newNode.nickName, refNeighbors);

      newRoute.addNeighborToTable(n);
      deviceNode d = NodeContainer.getNodeContainer()
        ↪ .
      getNode(n.getNickname());
      updateNewDevice(d, false, newRoute);
      GraphTableCollection.getGraphTableCollection().
      storeGraphTable(newRoute);
      isLeaf = true; // reset position

  else
    for(Neighbor n: newNode.returnNeighbors())
    deviceNode d = NodeContainer.getNode(n.
      ↪ getNickname());
    currTable.addNeighborToTable(n);
    updateNewDevice(d, isLeaf, currTable);
```

Listing 8.1: New device joining

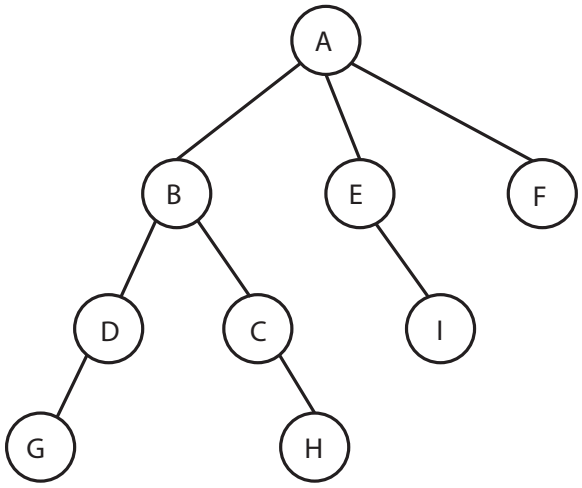

Figure 8.9: BFS tree

recursively call the method again to generate a list of the neighbours until we are at the Network Manager. As the next node is not a leaf, subsequent nodes will be appended to the already existing list instead of generating a new list. Once the current node is the Network Manager, the route is complete and is stored in the graph table container.

Finding shortest path

Once the device has joined the network and all paths are generated, we can calculate the most effective route. The Network Manager determine the most effective route by number of hops, as can be seen in listing 8.2.

Device failing or leaving

If a node fails or leaves the network, one of its neighbours will detect its absence through the keep-alive Data Link PDU's they periodically send between each other. The discovery will be reported to the Network Manager, which will update its routing tables and forward the change to all the affected nodes. To perform the updates, the manager will find all the stored routes and delete the ones containing the node. If a route is part of a shortest path (currently active routing path), a new route will have to be set as the active route. Taking node C in figure 8.8 as an example, the shortest route is calculated to be $A \rightarrow B \rightarrow C$ in figure 8.9. If node B fails, the shortest path to node C is deleted, and the route $A \rightarrow E \rightarrow C$ is set to shortest path. Code for this sequence can be seen in listing 8.3.

The failing node is found in the collection of nodes stored at the Network Manager, and deleted. If the node should reappear in the network, a new join process has to be initiated. We determined that this would be the appropriate way to go, as nodes will usually not move around much and nodes leaving will usually not reappear[5]. This saves

[5]Strong temporary electrial interference blocking communication is not taken into

```
findShortestRoute(deviceNode node){
  Collection<GraphTable> it =
  GraphTableCollection.getGraphTableCollection().
  returnTable().values();
  GraphTable best = null;
  int nJumps = 0;

  for(GraphTable t: it)
    if(t.returnDestNickName() == node.nickName)
     if(best == null)
       best = t;

     if(t.returnNeighbors().size() <
       best.returnNeighbors().size() || nJumps == 0)
       best = t;
```

Listing 8.2: Shortest route

```
updateDeviceFailing(deviceNode failingNode)
  NodeContainer.getNodeContainer().removeNode
  (failingNode.nickName);
  Collection<GraphTable> it =
   GraphTableCollection.getGraphTableCollection
  ().returnTable().values();
  Iterator<GraphTable> tableIterator = it.iterator();
  while(tableIterator.hasNext())
    GraphTable curr = tableIterator.next();
    ArrayList<Neighbor> neighs = curr.returnNeighbors
       ↪ ();

    for(Neighbor n: neighs)
      if(n.getNickname() == failingNode.nickName)
        tableIterator.remove();
        GraphTableCollection.getGraphTableCollection
           ↪ ().
        deleteTable(curr);
```

Listing 8.3: Update device

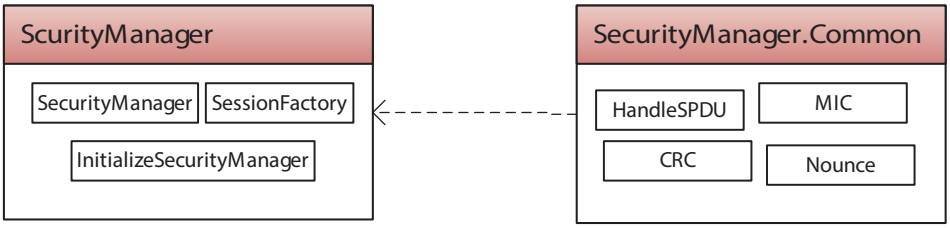

Figure 8.10: Security Manager Packet Structure

the Network Manager the trouble of storing temporary information which will have to be re-evaluated if the node should rejoin the network. After the node is deleted from the collection, all the graphs are parsed and checked if they contain the deleted node. If the node is part of a graph, the graph is now invalid and deleted.

8.5 Security Manager

A minimal Security Manager has been implemented in the project. The Security Manager packet structure is explained below and can be seen in figure 8.10.

8.5.1 Security Manager Packet Structure

SecurityManager - Contains the necessary values for keeping a session between two devices secure.

SecurityManager.Common - Contains the security functions implemented as separate modules. CRC, MIC and the Nounce are implemented here.

8.6 Access Points

The Access Point acts as a sink for traffic between the network and the Gateway. Asperheim et al. ran a single Access Point as part of the Gateway. We decided early on that one of the first tasks would be to separate the Access Point from the Gateway. The standard does allow the Access Point to be a part of the Gateway, but the former implementation would have to be changed in order to support more than one Access Point. The separation would allow us to create and run multiple Access Points, station them on separate locations, as well as running them on their own platforms. Once they had been physically separated, we were able to decide on a transmission method between the Access Points and the Gateway.
Considerations had been made in the previous project for making several Access Points as queue systems were implemented and intended to be

account

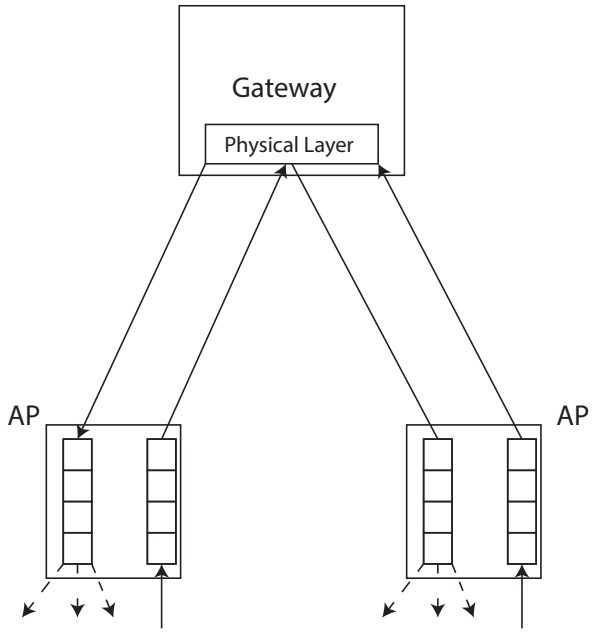

Figure 8.11: Message passing between Gateway and Access Points

shared between the devices. As part of the separation, we provided a separate TxQueue (transmit queue) and RxQueue (receive queue) to each running Access Point seen on figure 8.11. Previously, connection information such as neighbour and link tables were shared between every Access Point, making more than one Access Point redundant as they provided the same routes and functionality. The neighbour and link tables were restructured and every AP now contains their own tables as every node in the network.

8.6.1 Separating the Access Points

We decided one of our goals would be to separate the Access Point from the Network Manager to a running program of itself. This would help the Sensor Network greatly by adding more throughput and reliability, by providing alternative routes to the Gateway. We implemented two kind of Access Points; clock-providing and non clock-providing Access Points. The idea behind this is to enable Access Points over a Wireless LAN connection that could not accurately get the time from the Gateway, but would still increase throughput of the network. There must always be at least one clock-providing Access Point in the network.

For this to work we would have to synchronise clocks between the Gateway and the clock-providing Access Points. It would also require dividing the given code from the earlier project into two running programs, as well as finding a suitable solution for reliable traffic between the Gateway and the clock-providing Access Points.

Our solution in the end was to implement the Access Points close to how we implemented the nodes in the network. This was in part to save

development time, as it would minimise redundancy. But it was also an excellent way for us to use the security mechanisms in WirelessHART between the Access Point and Gateway. It is easy to implement a new connection protocol for the connection between Access Point and Gateway if it becomes necessary.

8.6.2 Clock Synchronisation

All the clocks in the network need to be synchronised in order for communication to work. Time Division Multiple Access, as discussed in section 2.1 on page 12, requires a close to perfectly synchronised network. The channel hopping further complicates the problem, as this also needs a very good clock synchronisation. The way we have implemented this is to use the Absolute Slot Number method as the standard describes. ASN is a 5 byte timer that starts at zero when the Gateway is initiated. Most packets contain either a full ASN or an ASN snippet to synchronise a device's clock to that of its neighbours. The ASN method also gives great possibility to synchronise commands over multiple devices in the network, such as collectively changing the Network Key.

To handle the timer in our implementation we have created two classes, *Timer* and *AbsoluteSlotNumber*, to handle all of the time keeping. The *AbsoluteSlotNumber* contains methods to set, get and synchronise time, as well as deciding what slot we are in a superframe. The *Timer* class contains all timers on the device, such as how often to send an advertisement or how often to retransmit un-acknowledged packets. This class also handles delayed commands as explained above.

8.6.3 Dividing the code

When we started, the Access Point were integrated into the Network Manager. This made it impossible to have more than one Access Point in the network. We started implementing this as a secondary program with a TCP connection between the Access Points and the Network Manager with some self-made commands like *SYNCHTIME, LEAVE, EXIT, DATA, PING, JOIN and LOG*. When we decided to start simulating the nodes instead of running this on real nodes we had to develop the whole WirelessHART stack. That opened the possibility to implement an Access Point in the same way we implemented a node. This would make the Access Points a lot smarter, and would make it easy to update the project in the future as all devices ran the same communication protocol.

We wanted to use TCP connections between the Access Points and the Network Manager, but due to time restrictions we had to down-prioritise this. To do implement this functionality, a new TCP-Connection class that implements the Connection interface has to be created and integrated into the Physical Layer.

8.7 Security

The WirelessHART standard implements security through a CCM* (Counter with CBC-MAC)[6] encryption algorithm defined in IEEE 802.15.4. Security is applied at the Data Link Layer for authentication and at the Network Layer for authentication and encryption. Our intention were to implement the security according to the standard, but realised that the amount of work would be too heavy due to the level of detail each implementation would require. The CCM* mode utilises a 128-bit AES encryption key for encrypting the data transmitted. In figure 8.12 we present the sequence of security for a transmission, description of each step below:

1 The Network Layer sends a packet to the encryption module.

2 The encryption module encrypts the packet with the key specified for the current transmission.

3 The encrypted packet is attached to the NPDU.

4 The NPDU is sent to the CRC calculation module.

5 The CRC module calculates a value for the NPDU to be transmitted. The value is put into the CRC field in the DLPDU.

6 The MIC of the DLPDU is calculated. The values of the header from *0x41* to the end of the payload is authenticated.

7 The Generated value is put into the MIC field in the header.

8 The DLPDU is sent to the Physical Layer and transmitted over the medium to a receiving device. At the receiving device, the DLL forwards the payload to the CRC calculation method.

9 The CRC module calculates the CRC value of the payload and compares it to the value in the CRC header field. If the values are equal, the message is passed to the Network Layer (9a). If not, it is discarded (9b).

10 The Network Layer strips off the headers, and sends the payload to a decryption method along with the key to decrypt it with.

11 The module decrypts the payload into cleartext.

12 The data is returned to the Network Layer and further processed.

8.7.1 Data Link Layer Security

The CCM* mode at the Data Link Layer does not provide encryption. Instead the ciphertext generated is simply the MIC (see section 4.5.7 on page 41). The MIC is attached to the plain-text to form the MAC payload. As a consequence, the CCM* is used to authenticate a received message or

[6]Technique for constructing message authentication code of a packet from a block cipher

acknowledgement. Once we receive a message at the Data Link Layer, we generate the MIC of the received message to check whether the message is authentic or not. Only authenticated messages are forwarded to the Network Layer, whereas messages failing the check are discarded.

8.7.2 Network Layer Security

The payload of the Network Layer is encrypted, opposed to the payload of the Data Link Layer which is simply authenticated. The transmitting device encrypts the message, and only once the packet arrives at the Network Layer at the destination device is the packet decrypted. As specified in the standard, intermediate devices will only forward the packet based on information provided in the Data Link Layer header. Even if a malicious device should intercept the packet and attempt to read the content, only the sender and receiver are able to decrypt the message due to the symmetric key[7]. As mentioned earlier, we decided to implement a simpler encryption algorithm than the CCM*. We still wanted to provide the authentication and encryption properties of CCM*; both for still maintaining a realistic use of the algorithm and the possibility of expansion as future work. We did some research and found existing security libraries in java called *javax.crypto.Cipher* and *javax.crypto.spec.SecretKeySpec*. These libraries provided us with the functionality of cryptographic ciphers for encryption and decryption [12].

Transmitting side

Listing 8.4 presents the implementation providing encryption of a plaintext for the Network Layer.

```
1  public byte[] encrypt(String key, byte[] payload){
2          byte[] encrypted = null;
3          byte[] raw = key.getBytes();
4          SecretKeySpec skeySpec = new SecretKeySpec(
               ↪ raw, "AES");
5
6          try{
7              Cipher cipher = Cipher.getInstance("AES")
                   ↪ ;
8              cipher.init(Cipher.ENCRYPT_MODE, skeySpec
                   ↪ ); // set mode along with byte key
                   ↪ and algorithm
9              encrypted = cipher.doFinal(payload);
10             return encrypted;
11         }
12         catch(Exception e){
13             e.printStackTrace();
14         }
```

[7] encryption key shared between two parts

```
15          return null;
16      }
```

Listing 8.4: General encryption algorithm

The method takes an encryption key and the payload as input. The key is first converted to raw bytes, before a *SecretKeySpec* object is generated by determining the raw byte key and the algorithm to be applied. We decided to use the AES encryption key for a few reasons; it is a secure algorithm and WHART's encryption use AES-128 in combination with additional security functions. The encryption key used within listing 8.4 are checked to be under 128 bits before calling on the method, which ensures AES-128 as keys lower than 128 bits are padded. A *Cipher* object is created with AES encryption, and initialised with mode set to encrypt and the secret key. The payload is encrypted using the cipher, and returned afterwards.

Receiving side

At the receiving end the process is close to reversed. Once a packet is received at the Network Layer, we check the four least significant bits of the security control byte by shifting the remaining bits out. The four bits determine which security type is utilised, and what type of key we need to use to decrypt the packet. The same method seen in listing 8.4 is used, except for line eight which is changed to:

```
1 cipher.init(Cipher.DECRYPT_MODE, skeySpec);
```

The security mode is now set to decrypt, and the module will get the key corresponding to the value set in the security control byte before decrypting the message and passing it on to the Transport Layer. The standard does not specify which values in the control byte correspond to the different keys, as it only refers the user to check a HCF Enumeration table[8]. We made it general and implemented it as *0 - Session Key, 1 - Join Key, 2 - Handheld Key*. We used the Join Key for communication between the Network Manager and a joining device until full integration was completed. Once full integration was accomplished, a Session Key was generated at the Network Manager and distributed to the node encrypted by the Join Key. Safe distribution of the key was accomplished, and the session key can now be used for encryption. The case for encrypting handheld devices were created, but not implemented as we had no such devices.

8.8 Packet Construction and transmission

8.8.1 Implementation of the Protocol Data Units

The Protocol Data Units are the headers and tails of the different layers. There are 3 different main PDUs together with 5 sub-layer PDUs, all of

[8] available for purchase from HART Communication Foundation

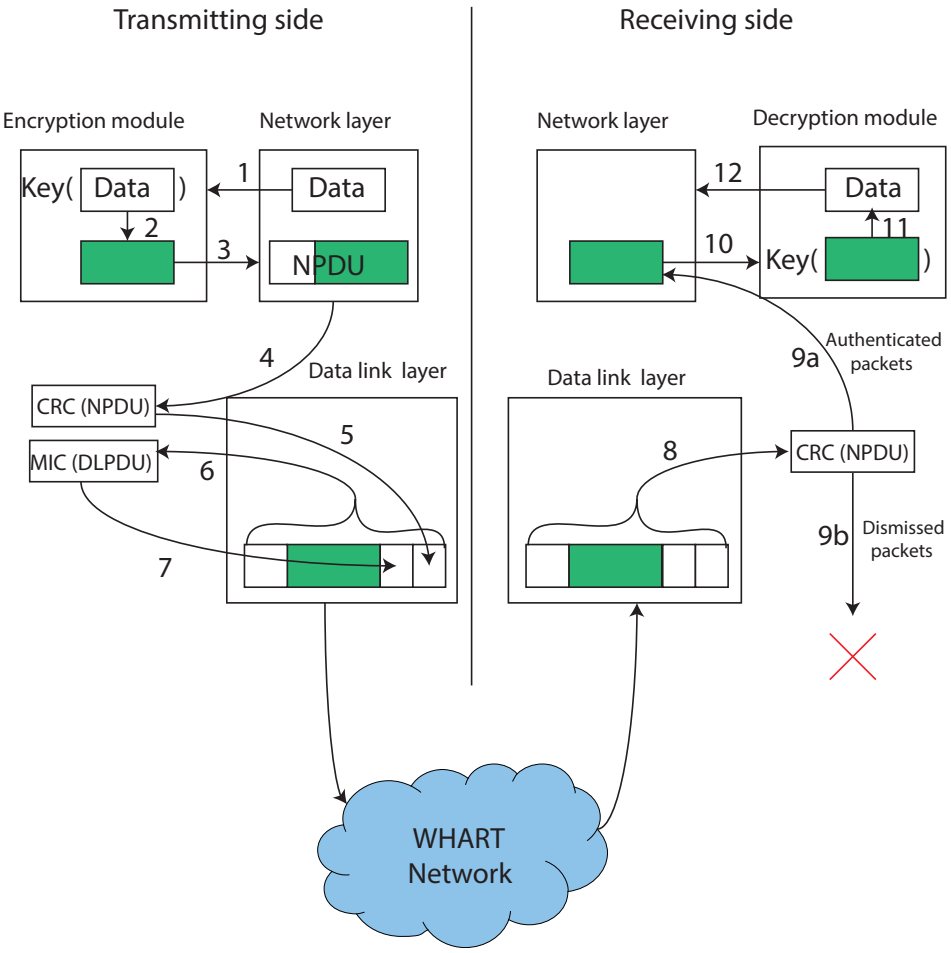

Figure 8.12: Security functions

which are implemented in our solution and are described throughout chapter 4. The PDUs got two constructors, one for constructing the object, and one for unmarshalling[9] the object, as can be seen on line 6 and 7 of listing 8.5. The method toByteArray() on line 9 will marshal the object into a series of bytes that can be used to send over a socket. Here we could use Java's *Serializable interface*, but as this would create an XML[10] based data structure it would not create a byte stream equal to that of the standard. Therefore it would not be possible for a proprietary node to read the data, so we had to manually create the methods for marshalling and unmarshalling the objects. A full implementation of the Data-Link Protocol Data Unit can be found in appendix E.1.

```
1  public interface PDU {
2      /* ******************* *
3      * Enums and Variables *
4      * ******************* */
5      public PDU(byte[] data) ;
6      public PDU(/*VARIABLES*/) ;
7      public byte[] toByteArray() ;
8      /* ******* *
9      * Getters *
10     * ******* */
11 }
```

Listing 8.5: PDU class skeleton

8.8.2 Data Link Protocol Data Unit

The following section presents the DLPDU construction implemented.

Cyclic Redundancy Check

WirelessHART uses Cyclic Redundancy Check based on the 16 bit *ITU-T CRC polynomial (CRC16)*. CRC is used to detect bit errors occurring at the Data Link Layer payload during transmission. At the receiving side, a value is generated of the DLL payload and compared to the value stored in the CRC field at the header. If the values match, the transmission has been error free. The CRC is calculated using the polynomial displayed in equation 8.1, equalling *0x1021* in hex.

$$G_{16}(x) = x^{16} + x^{12} + x^5 + 1 \tag{8.1}$$

In order to calculate CRC, a set of values are needed. We solved this by creating a table of 256 values as seen in listing 8.6. The table is filled by parsing over every position, assign a value to the position and again run over this field eight times while shifting the values based on the position of the table, in combination with the polynomial value.

[9] Extracting a data structure from a series of bytes
[10] Extensible Markup Language - a format that is both human- and machine-readable

```
// values needed for CRC
    private static final int poly = 0x1021;
    private static final int[] table = new int[256];

    // fill CRC table
    static{
        for(int i = 0; i < 256; i++){
            int crc = i << 8;
            for(int j = 0; j < 8; j++){
                if((crc & 0x8000) == 0x8000)
                    crc = (crc << 1) ^ poly;
                else
                    crc = (crc << 1);
            }
            table[i] = crc & 0xffff;
        }
    }
```

Listing 8.6: CRC initialization

```
public int generateCRC(int crc, byte[] bytes, int off
    ↪ , int len){
        for(int i = off; i < (off + len); i++){
            int b = (bytes[i] & 0xff);
            crc = (table[((crc >> 8) & 0xff) ^ b] ^ (
                ↪ crc << 8)) & 0xffff;
        }
        return crc;
}
```

Listing 8.7: CRC calculation

The table generation is performed the same way on every device, ensuring the devices have the same tables when generating and checking the CRC values. Listing 8.7 shows how we calculated the CRC of a packet. The module takes arguments *CRC* which determines the value into the table, the *payload*, the *offset* which *CRC* is calculated from and the *length* of the payload. We loop through the payload byte by byte, applying a bitwise AND operation, before using the table as a lookup to calculate the final value which is returned. If the generated CRC corresponds to the value stored within the header, the packet is acknowledged and sent to the Network Layer.

Message Integrity Code

Keyed Message Integrity Code (MIC) is a technique used at the Data Link Layer for authenticating packets. The process is necessary to determine if

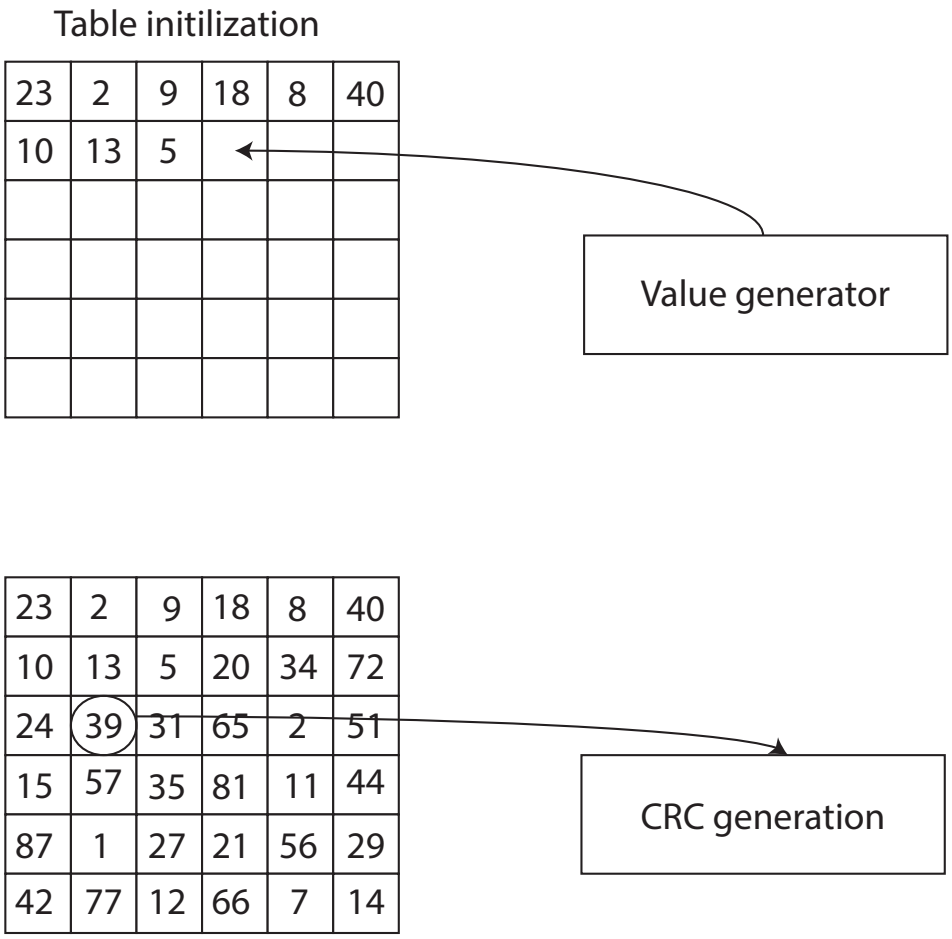

Figure 8.13: CRC-16

the receiving packet is coming from an approved device. Only packets that are successfully authenticated are processed and if necessary, responded to. To provide MIC, the protocol uses CCM* mode combined with AES-128 block cipher, both introduced in section 8.7. The DLPDU is not enciphered, but instead the content is authenticated with the four-byte MIC. To successfully generate a message integrity code, four values are needed:

- Data to be authenticated but not enciphered.
- The message to be enciphered.
- A 13-byte nounce, following it the next section.
- The 128-bit AES key used for encryption.

We started implementing a CCM* encryption method for the MIC, but discovered that the level of detail required for a correct implementation were too heavy. The method in itself is a few hundred lines of advanced code, which would require too much time considering what we would gain from implementing it. As we already had implemented an encryption algorithm at the Network Layer, we decided to skip the CCM* calculation for now and focus on other, more important areas.

Nounce

The nounce calculation is dependant on the addressing mode used for the current packet. Independent of the addressing mode, position zero to four is populated by the current *ASN*. If the packet is part of communication with a device not yet fully integrated into the network, the *EUI-64 address* is used. To determine how the *nounce* is generated, we perform a simple test to determine the addressing scheme employed in the packet by the Network Layer. If it is a *EUI-64 address*, we put values *0x00, 0x1b, 0x1E* into position five through seven in the nounce. Position eight and nine are populated by the *expanded device type code*, and 10 through 12 are assigned to *device ID*. If nicknames addressing is used, we fill position five to 10 with *0x00* and place the nickname in position 11 to 12.

8.8.3 Network Protocol Data Unit

The Network Protocol Data Unit ensures end-to-end communication in the network. We implemented this according to the standard, described in section 4.10 on page 44.

8.8.4 Transport Protocol Data Unit

The Transport Protocol Data Unit is handled in the Application Layer of our Implementation. It is implemented according to the standard, described in section 4.6.4 on page 46. The TPDU contains commands sent between devices.

Command Based Communication

The Application Layer of the WirelessHART stack is command driven. A Transport Layer PDU (Section 4.6.4) is constructed of a number of commands separated into different commands using a *Parser* and a *CommandHandler*. The *CommandProcessor* then processes the commands by calling the *executeCommand()* method inside the command. All commands extend the abstract class *Command* as can be seen in listing 8.8. The enum *CommandName* contains the command's command number, the size of the marshalled object and ways to get *commandName* from the *commandNumber*. In the implementation each command is an object, much like the PDUs described in section 8.8.1. It contains constructors depending on the differences between request and response packets, a *toByteArray()* method together with the *executeCommand()* and *generateResponse()* methods. When a node in the network requests a service it sends a Request Command to the node. This usually look the same as the Response Command, but can be different. Appendix B describes the variables needed for the *Request* and *Response* of each implemented command together with the specific *Response Codes* the command uses. We have implemented the commands as close to the standard as we can, but since we do not have the HCF Enumeration tables we had to make educated guesses on some of the bytes. We have only implemented the Response Codes needed in order to make the network work, but it should be easy to extend if needed.

```
1  public abstract class Command {
2      CommandName commandName ;
3      byte length ;
4      long executionTimeOfCommand = 0l ;
5
6      ArrayList<ResponseCode> responses = new
          ↪ ArrayList<ResponseCode>() ;
7
8      protected Command(CommandName commandName) {
9          this.commandName = commandName ;
10     }
11
12     public CommandName getCommand() {
13         return commandName ;
14     }
15
16     public short getCommandNumber() {
17         return commandName.getCommandNumber() ;
18     }
19
20     public long getExecutionTimeOfCommand() {
21         return executionTimeOfCommand ;
22     }
23
24     public void setExecutionTimeOfCommand(long
```

```
        ↪ executionTimeOfCommand) {
25          this.executionTimeOfCommand =
                ↪ executionTimeOfCommand ;
26      }
27
28      public abstract byte[] toByteArray() ;
29      public abstract ResponseCode executeCommand() ;
30
31      public void addResponseCode(ResponseCode
            ↪ responseCode) {
32          responses.add(responseCode) ;
33      }
34
35      public ArrayList<ResponseCode>
            ↪ getAllResponseCodes() {
36          return responses ;
37      }
38
39      public abstract byte[] generateResponse() ;
40  }
```

Listing 8.8: abstract class Command

8.9 Chapter Summary

This chapter has looked at the work and changes made on the code. The changes to program structure were first presented, followed by the implementation of each network device in sections 8.2 to 8.6. At the end of the chapter, security functionality applied to a packet transmission were detailed in section 8.7, followed by DLPDU construction and transmission in section 8.8.

Part IV

Review

Chapter 9

Testing and Evaluation

The following chapter presents the reader with the test environment and test results obtained when running the implementation. Section 9.1 details the test network established and assumptions made when configuring the network. Section 9.2 provides an overview of the test results obtained. Section 9.3 evaluates the implementation of each device, followed by sections 9.4 and 9.5 evaluating each result against the requirements determined in chapter 5 on page 63. The chapter concludes with a review of the development process.

9.1 Test Bed

In order to test the implementation and how devices act, we created a test network consisting of nine devices. To specify communication and device roles we created a file containing the Join Key and Network ID, usually provided to a new device by a hand-held device. It also holds connection information as can be seen in table 9.2. Lastly it contain the device's *Unique ID*, a 8 byte address as per the standard.

Since we simulate a wireless environment on a wired network, we had to give the nodes information about their neighbouring devices. We used UDP[1] multicast in order to simulate that all devices in range received all information[2].

For all this to work we had to create a connection module. We currently have a fully developed UDP module that we use for testing and simulating devices. We have also started work on a TCP module that we wanted to use between the Network Manager and the Access Points in the real network.

The UDP module implements channels by adding a one byte header containing the channel it was sent on. When a device receives this packet, it compares the sent on channel with its current channel. If these are different, the packet is discarded. We have also implemented methods for packet-loss and packet-distorting.

We had three computers acting as hosts for the devices:

[1] User Datagram Protocol - A stateless transmission protocol

[2] As long as they are listening on the correct channel

- A server running an Intel Atom 1.8 Gigahertz Dual Core Central Processing Unit and 4 gigabytes of Random Access Memory. This device is nicknamed Server.
- Two Raspberry Pi computers described in section 7.1.4. These are nicknamed PI1 and PI2.

Name	IP	Devices on machine
Server	192.168.1.100	NM, AP1, N1
PI 1	192.168.1.101	N2, N3, N6
PI 2	192.168.1.102	AP2, N4, N5

Table 9.1: Computers and Devices

These three computers ran the nine devices needed to test our implementation. It consists of one Network Manager, two Access Points and six Nodes. The nodes were spread on the three computers as can be seen on in table 9.1. An easier view on how the nodes were connected can be found in figure 9.1.

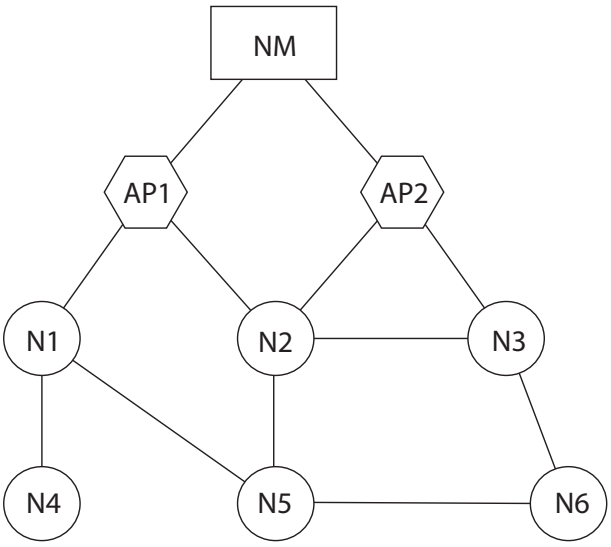

Figure 9.1: Test Bed Network

We created a superframe with 30 slots. We then added a discovery, join and broadcast link. This meant that every three seconds the superframe would repeat itself.

All devices in the network run the software described in chapter 8.

9.1.1 Nodes

The nodes were connected in a topology as can be seen in figure 9.1. Addresses and ports for the different nodes can be found in table 9.2.

Name	Long Address	Computer	Port	Connects to
NM	0xF980 0000 001	Server	4450	localhost:4451, 192.168.102:4452
AP 1	0x001B 1EEA 2000 0001	Server	4451	localhost:4453, 192.168.1.101:4454
AP 2	0x001B 1EEA 2000 0002	Pi 2	4452	192.168.1.101:4454, 192.168.1.101:4455
Node 1	0x001B 1EEA 3000 0001	Server	4453	192.168.1.102:4456, 192.168.1.102:4457
Node 2	0x001B 1EEA 3000 0002	Pi 1	4454	localhost:4455, 192.168.1.102:4457
Node 3	0x001B 1EEA 3000 0003	Pi 1	4455	localhost:4458
Node 4	0x001B 1EEA 3000 0004	Pi 2	4456	192.168.1.100:4453
Node 5	0x001B 1EEA 3000 0005	Pi 2	4457	192.168.1.101:4458
Node 6	0x001B 1EEA 3000 0006	Pi 1	4458	localhost:4455

Table 9.2: Test bed set-up

9.2 Test Results Overview

This section presents our results from the tests we performed at the end of the development cycle. During the development we actively used Unit Tests in order to test methods, modules and mechanisms as we developed them. The tests gave for the most part true or false outputs and is not mentioned in this section.

9.2.1 Synchronising to the network

Once a node is switched into JOINING mode the node starts to listen for *AdvertisementDLPDUs*. These packets are sent from all nodes where *ADVERTISEMENT_INTERVAL_TIMER* != -1, meaning it will send an advertisement every *ADVERTISEMENT_INTERVAL_TIMER* milliseconds. In our tests this was 3000 ms.

A device that is not synced to the network uses the *CHANNEL_SEARCH_TIME*[3] to switch channels while a synced device uses the algorithm

$$channelNumber = (channelOffset + ()ASN\%numberOfActiveChannels) \tag{9.1}$$

This means actually hitting the correct channel at the exact time frame of which the *AdvertisementDLPDU* is sent is based on luck as can be seen in figure 9.2.

There are ways to battle this. We could make a superframe to an advertising device that only was broadcast and join links. Another way to maximise the chance to hit the correct time frame is to blacklist as many

[3]Default value is 300 milliseconds

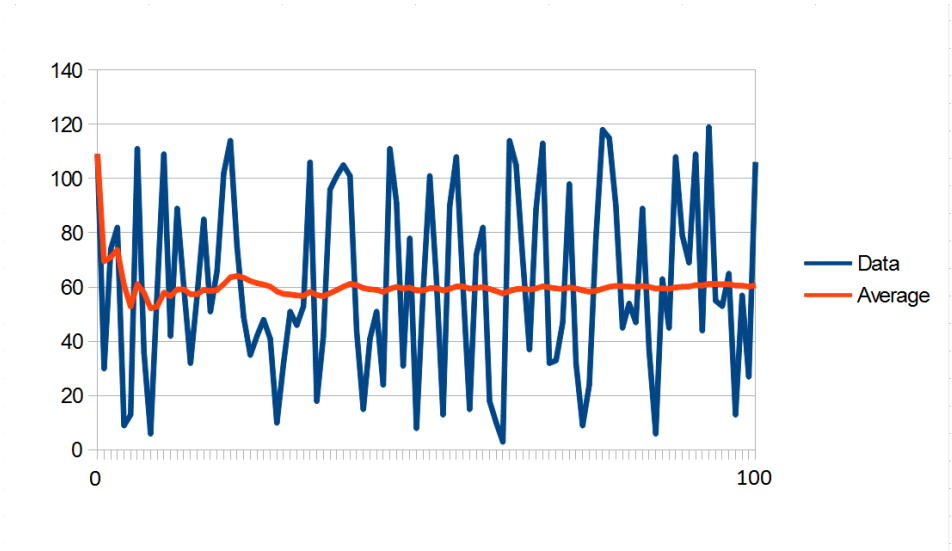

Figure 9.2: Average synchronisation time of a node

channels as possible. Both these options have their drawbacks as the first option will use a lot of bandwidth and battery on the advertising device. The second option makes the network more vulnerable to jamming, both incidental and deliberate.

The test is based on simulations and self made restrictions of which may be error prone and could give results that would not be seen in a real WirelessHART network.

We found one glitch during testing where the device synchronised to the network, but was always one channel ahead or behind the sender. This make it impossible for the device to hear any traffic as it thinks it is synchronised. The timer is so far behind that it listens to another channel, even though it is inside the correct time frame.

9.2.2 Joining the network

To get a device to join the network proved to be one of the most challenging requirements during the project. It is a complex routine as illustrated in figure 9.3.

The node needs to synchronise to the network as described earlier, then collect as much information it could before sending a join request. The node collects this information by sniffing for traffic, reading header information as well as getting *AdvertisementDLPDUs*. After a *ADVERTISEMENT_WAIT_TIMEOUT*[4] and a *random wait timer*[5], the node sends the join request to one of the neighbours it had found during the sniffing period. This request needs to be sent on a join link. It is important for this node to listen to all join links in the superframe, as the Network Manager will

[4]Default value is 30 seconds
[5]This timer is 0 - 120 seconds

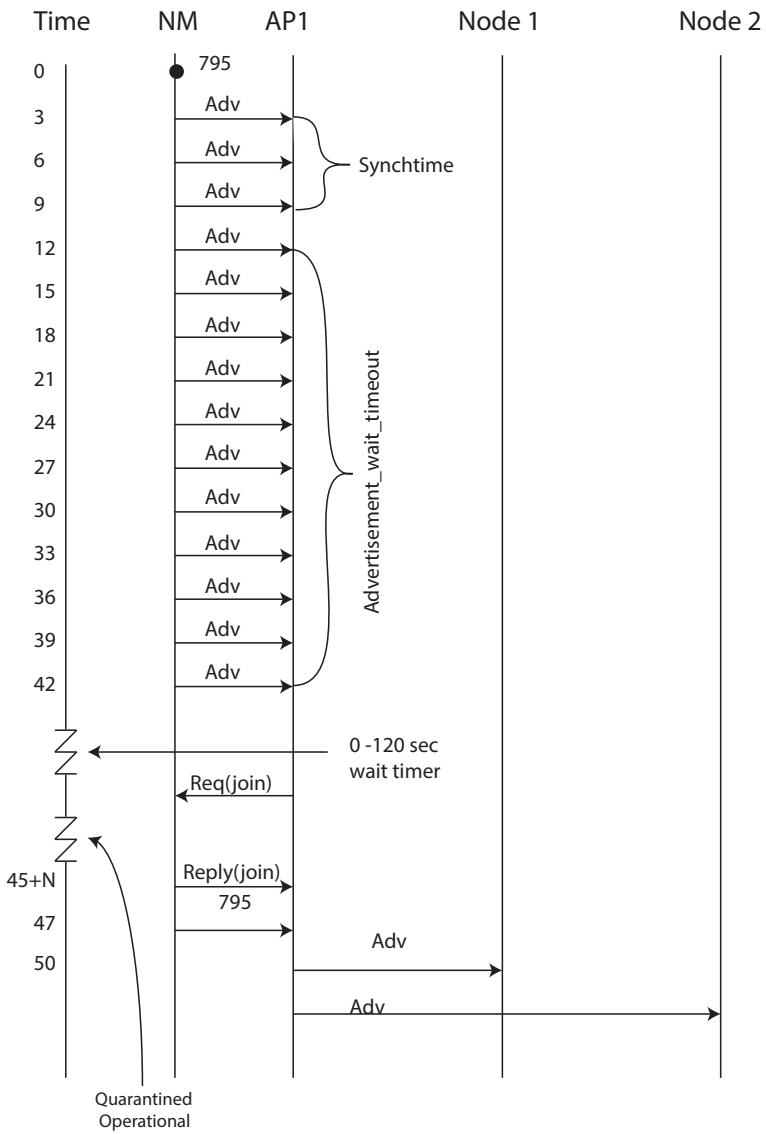

Figure 9.3: Node Join Sequence Diagram

send the join reply on the link it finds most suited based on variables such as signal strength, traffic paths and so on.

The Network Manager now receives the join request and uses it to calculate data, the node's parents, give a nickname, the Network Key and a session for further communication. The Network Manager sends a join reply to the designated parent of the joining node. The parent reads the packet and finds the proxy bit in the Network Layer Protocol Data Unit set, meaning it should send the packet on its join link.

When the joining node gets the join reply it enters a quarantined state where it has successfully joined the network, but only got security clearance to communicate with the Network Manager. For the node to become operational it needs to be provisioned with superframes, graphs and links from the Network Manager. Until then all communication need

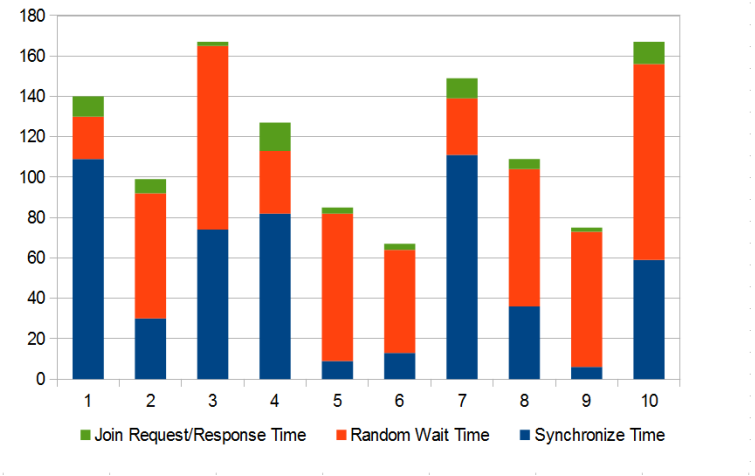

Figure 9.4: Time for a device to go from JOINING to QUARANTINED state

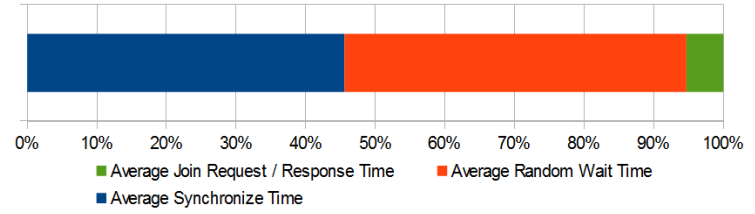

Figure 9.5: Average time for a device to go from JOINING to QUARANTINED state

to be done by the parent proxy. The time used for a node to go from joining state to quarantined is shown in figure 9.4 and 9.5.

Improvements to the Joining Sequence

The join sequence is slowed down considerably by the random wait timer. A way to remove the need for this timer is to implement two new commands; *Clear to Join* and *Ready to Join*. These commands will act exactly as the *Clear to Send* and *Ready To Send* control frames in *Carrier Sense Multiple Access* described in section 2.2 on page 13.

When a joining node is ready to join it sends a *Clear to Join* command on a join link. A node already in operational state then sends a *Ready to Join* command back to the node.

9.2.3 Security

There are three security mechanisms implemented in WirelessHART. Message Integrity Check, Cyclic Redundancy Check and Encryption. These mechanisms are described in detail in section 8.7. A short description of these mechanisms is presented in the following sub sections before evaluating our implementation.

Message Integrity Check

Message Integrity Check is a way to authenticate the sender of the packet. It is used both in the Data Link Layer for hop-to-hop authentication, and in the Network Layer for end-to-end authentication. As described in section 8.8.2, MIC uses a CCM* based encryption technique for authenticating packets. As we only partly implemented the encryption, we did not apply the algorithm to packets sent between the devices. Our implementation of the algorithm can be seen in appendix E.3 on page 172.

Cyclic Redundancy Check

Cyclic Redundancy Check is implemented on the Data Link Layer for detecting errors in the packet. This implementation can not be used to correct bit errors in the packet, only detect the errors. If a bit error is detected, the receiver discards the packet and the sender will have to retransmit the packet. The value *max_reply_time* currently set to 30 seconds, is used at the Network Layer to trigger retransmissions. In listing 9.1 we demonstrate the calculation implementation. A payload with the text "*Device 0x23354 has discovered a leak in gasline 2!*" is calculated a CRC value for, resulting in the value 56359 by starting at position 0 in the CRC table. We send the message to four different devices, each receiving the payload differently.

Node 1 Verifies that the correct packet is received as calculated CRC value corresponds to the CRC field in the packet header.

Node 2 Determines that an incorrect CRC value is generated. In the example, we have started at table position one instead of zero, which results in a large difference in the CRC values.

Node 3 CRC value is incorrect. Table position is set to 98 instead of zero. A larger differential error occurred in case 2, which means that there is no linear connection between CRC value and starting table position.

Node 4 - Received a packet where *0x23354* has been modified to *0x23359* during transmission. The CRC value generated is wrong, and the packet is discarded.

```
1 Payload sent: 'Device 0x23354 has discovered a leak
    ↪ in gasline 2!'
2 CRC value generated is '56359' using 0 as the
    ↪ starting position in the table
3
4 ---Receiving side node 1---
5 Payload received is 'Device 0x23354 has discovered a
    ↪ leak in gasline 2!'
6 CRC value generated is '56359' using 0 as the
    ↪ starting position in the table
7 Generated CRC value is correct
```

```
8
9  ---Receiving side node 2---
10 Payload received is 'Device 0x23354 has discovered a
     ↪ leak in gasline 2!'
11 CRC value generated is '24899' using 1 as the
     ↪ starting position in the table
12 Generated CRC value is incorrect. Bit-errors exist in
     ↪  packet or table position
13
14 ---Receiving side node 3---
15 Payload received is 'Device 0x23354 has discovered a
     ↪ leak in gasline 2!'
16 CRC value generated is '62485' using 98 as the
     ↪ starting position in the table
17 Generated CRC value is incorrect. Bit-errors exist in
     ↪  packet or table position
18
19 ---Receiving side node 4---
20 Payload received is 'Device 0x23359 has discovered a
     ↪ leak in gasline 2!'
21 CRC value generated is '7068' using 0 as the starting
     ↪  position in the table
22 Generated CRC value is incorrect. Bit-errors exist in
     ↪  packet or table position
```

Listing 9.1: CRC calculation results

Message Encryption

The encryption of the message is used by the Security sub-layer. It encrypts the Network Protocol Data Unit payload with one of the following keys: *session*, *handheld* or *join key*. The module takes a key and the payload as arguments, and attempts to decrypt the payload with the key. In listing 9.2, we show encryption of a packet and several decryptions. The text *"This is a packet transmission to test encryption"* is encrypted and sent to several devices. Each of the cases are elaborated on below:

Sender 1 - We encrypt the packet with the key *"ThisIsCorrectKey"* and transmit it.

Node 1 - Decrypts with the correct key, and correct value is extracted.

Node 2 - Decrypts with the wrong key, and an error message is returned. The algorithm throws an exception as the returned value cannot be compared to a value in the utf-8 table. We print an error message, and discard the packet.

Sender 2 - The packet is encrypted with the "wrong" key and transmitted.

Node 3 - We attempt to decrypt the message with the correct key. As the message was encrypted with the wrong key, the same case as for Node 2 happens. This is also the case for when a malicious node intercepts a message and attempts to decrypt it with a wrong key.

```
1  The payload sent: 'This is a packet transmission to
   ↪ test encryption'
2  payload is encrypted with key: ThisIsCorrectKey
3  encrypted value is: 53  EB  6E  A6  CB  4B  7E  A5
   ↪ D4  F0  F0  42  31  DB  F0  82  34  D4  84  0C
   ↪  14  28  89  32  A1  B0  07  DA  34  58  3B  C6
   ↪   5F  56  BB  87  42  89  79  1D  37  74  AB
   ↪ 20  E8  F4  6E  22  FD  BB  0A  58  51  6B  B6
   ↪  49  5D  5A  81  70  8E  F9  C6  7F
4
5  ---Receiving side node 1---
6  Received encrypted value: 53  EB  6E  A6  CB  4B  7E
   ↪  A5  D4  F0  F0  42  31  DB  F0  82  34  D4  84
   ↪   0C  14  28  89  32  A1  B0  07  DA  34  58  3
   ↪ B  C6  5F  56  BB  87  42  89  79  1D  37  74
   ↪ AB  20  E8  F4  6E  22  FD  BB  0A  58  51  6B
   ↪  B6  49  5D  5A  81  70  8E  F9  C6  7F
7  Decrypting with key: ThisIsCorrectKey
8  Gives 'This is a packet transmission to test
   ↪ encryption'
9
10 ---Receiving side node 2---
11 Received encrypted value: 53  EB  6E  A6  CB  4B  7E
   ↪  A5  D4  F0  F0  42  31  DB  F0  82  34  D4  84
   ↪   0C  14  28  89  32  A1  B0  07  DA  34  58  3
   ↪ B  C6  5F  56  BB  87  42  89  79  1D  37  74
   ↪ AB  20  E8  F4  6E  22  FD  BB  0A  58  51  6B
   ↪  B6  49  5D  5A  81  70  8E  F9  C6  7F
12 Decrypting with key: ThisIsWrongKey88
13 Decryption failed due to wrong decryption key
14
15 ---Sending side 2---
16 The payload sent: 'This is a packet transmission to
   ↪ test encryption'
17 payload is encrypted with key: ThisIsWrongKey88
18 encrypted value is: 53  EB  6E  A6  CB  4B  7E  A5
   ↪ D4  F0  F0  42  31  DB  F0  82  34  D4  84  0C
   ↪  14  28  89  32  A1  B0  07  DA  34  58  3B  C6
   ↪   5F  56  BB  87  42  89  79  1D  37  74  AB
   ↪ 20  E8  F4  6E  22  FD  BB  0A  58  51  6B  B6
   ↪  49  5D  5A  81  70  8E  F9  C6  7F
19
20 ---Receiving side node 3---
```

```
21 Received encrypted value: 1E 69 D4 5E C1 A9 EC
   ↪ 98 30 AD 9A A0 66 99 31 E6 AB 97 A5
   ↪ CC 0F DB BE 43 1A 40 2C 2C 47 4E 7
   ↪ A FA C4 DD 3E 7B 17 1E 53 9D 83 05
   ↪ 93 94 52 79 4E 3C 62 B3 40 3D 1B 3B
   ↪ 5A 5B FB E8 9F 62 D8 55 3A 47
22 Decrypting with key: ThisIsCorrectKey
23 Decryption failed due to wrong decryption key
```

Listing 9.2: Encryption results

9.3 Evaluation of the Implementation

In the following sections we evaluate the implementation performed during the project. First the different layers are discussed before Gateway, Network Manager and Security Manager are evaluated.

9.3.1 Evaluation of the Layer Implementation

In this section we evaluate the implementation of the layer structure. Our project introduced a layered architecture according to the standard, explained in the chapters WirelessHART at page 23 and implementation at page 87. We used most of our time on the Data Link Layer and the Application Layer. A minimal Physical and Network Layer was also implemented.

Physical Layer

We implemented a minimal Physical Layer based on an UDP[6] connection. The connector is easily replaced with a 802.15.4 connector by replacing UDPConnection.java with another file that implements the Connection interface and replacing a few lines of code. This layer does require more work, but as we worked in a simulation environment we could not implement and test it as we wanted.

Data Link Layer

Most of our efforts went to implement this layer. Most of the simulation is abstracted away in the physical layer, resulting in that the rest of the layers should work on an implementation towards a real network. We have implemented a fully working *Message Handling* and *Timer* module as well as a mostly working *Link, Superframe, Neighbour* and *Graph Table*. A message queue and retransmit queues are partially implemented. We have also implemented a working State Machine to send in the correct time windows.

[6]User Datagram Protocol - A stateless transmission protocol

We have implemented all Protocol Data Units needed for the Data Link Layer to operate. These PDUs carry data, acknowledgements, keep-alives, advertisements etc. as explained in section 4.5.6 on page 38.

We feel we have developed this layer according to the standard. There are however some implications as we did not have the enumeration tables that needed to be acquired from the HART foundation as seen in table C.1. Therefore we had to make some educated guesses on what different bits meant.

Network Layer

The Network Layer was the layer were we thought we would do most of our work at the start of this project. However, it did not turn out as we wanted due to problems with the code handed to us and with the hardware in use. We needed to implement a new Physical and Data Link Layer in order to start working on this layer which took more time than anticipated.

We implemented a minimal Network Layer that managed some basic routing. We also partially implemented the Session Table, but this is not in use as the implementation of the Security Manager is minimal. Queues for incoming and outgoing packets are implemented as well as a packet re-transmitter handled by the Timer in the Data Link Layer.

The PDUs in the Network Layer was implemented. The Network Layer PDU is handled as well as the Security Sub Layer PDU. We decided on handling the Transport Layer PDU in the Application Layer because it made more sense.

There are some future work to be done on this layer, as listed in chapter 11 on page 137.

Application Layer

The Application Layer is the most complete layer of this project. We implemented a *Parser* that split the Transport Layer PDU into commands and sends them to a *Command Processor*. The *Command Processor* then decides what commands were sent and runs the *executeCommand* method inside the command. We decided to implement commands in the same way we implemented the PDUs since we felt it is easier to work on objects rather than a raw byte string.

We implemented a basic set of commands as can be seen in appendix B. We did not implement the response codes however, as we did not need this to get our results.

9.3.2 Evaluation of the Gateway Implementation

The Gateway implementation inherited by the last students were virtually non existent as they focused their work on the nodes that we decided to discontinue. We therefore had to do a lot of work to make it work. We feel we have built a good foundation to continue working on, even though some core features are partially or not implemented.

Access Points

We now have an implementation that allows multiple Access Points. All devices in the network communicate using the same layered structure evaluated in the last section. This means that an Access Point needs to go through the same Join Process. We turned off all except one channel here so Channel Hopping was disabled. This made joining the network a lot faster. It is easy to add a new connection to the Physical Layer if we want a TCP[7] connection instead. We decided not to use too much time on implementing multiple connection possibilities as we felt it was not necessary for our thesis.

Network Manager

The Network Manager determines the logic within the network. It sets up routing tables, manages joining nodes, sets up superframes etc. Implementing a fully working Network Manager is in itself a thesis. We managed to create some basic routing tables, let devices join the network and set up basic superframes. There is no QoS and limited failure handling implemented.

Security Manager

To implement a working join procedure we had to do some work on the Security Manager. We created a SessionFactory for creating 128-bit keys for AES[8] encryption. We also added ways to generate and check MIC and CRC as mentioned earlier in the chapter. The Security Manager also manages Join-, Network- and the Well-Known-Key as discussed in chapter 8.

9.4 Review of Functional requirements

This section evaluates our test results and implementation against the requirements discussed in chapter 5 on page 63.

9.4.1 FR01 - Proper Advertise

Proper Advertise is implemented in the Data Link Layer. We have a *ADVERTISEMENT_INTERVAL_TIMER* that can be turned on by sending command *795 Write Timer Interval*[9] to the device that should start advertising. The device will then advertise every N milliseconds, as defined in the command. We feel this requirement is fulfilled and is working according to the standard.

[7] Transmission Control Protocol - provides reliable, ordered, error-checked delivery
[8] Advanced Encryption Standard
[9] Command 795 is described in appendix B.3

9.4.2 FR02 - Join Sequence

The joining sequence is tested and discussed in 9.2.2. The join request and join response is correctly propagated to the target device through the parent proxy as described in the standard. We feel this requirement is fulfilled according to the standard. We did not, however, implement any improvements to the join sequence due to time constraints. One possible improvement is described in section 9.2.2.

9.4.3 FR03 - Send pattern

The correct send pattern is implemented as per the standard. Correct headers are added in each of the layers and are evaluated in section 9.3.1.

9.4.4 FR04 - Functioning routing algorithm

A working routing algorithm is in place and described in detail in section 8.4.5 on page 97. The Network Manager uses a special Node Table that holds the whole topology of the network to create paths. To get the routing to nodes command *974 Write Route*[10] is used to supply a node with graph routing and 976[11] to supply source routing information.

We feel we got as far as we could with the routing between devices. We had more low-layer implementation to do than we had expected and did not have enough time to fulfil this requirement the way we intended at the start of the project.

9.4.5 FR05 and FR06 - Time Synchronisation and Time Division Multiple Access

Our implementation has implemented a time synchronisation module that is tested in section 9.2.1. We did encounter problems with time synchronisation and channel hopping as explained earlier. Time synchronisation is a complex thing to achieve in a wireless environment. We feel with the time and expertise we had during this project, that the requirement has been fulfilled. There are however improvements to be made here to further increase the accuracy of the synchronisation.

9.4.6 FR07 - Apply MIC and CRC

In section 9.2.3 the test results from running our implementation were presented. We demonstrated how the CRC algorithm calculated the value and compared the generated value with the value contained in the packet header. We also presented how the algorithm discovered bit errors in the packet, and that a small error resulted in a major change in the CRC value. The algorithm discovered the errors we introduced and fulfilled the requirements of a cyclic redundancy check. We determine this requirement

[10]Command 974 is described in appendix B.3

[11]Command 976 is described in appendix B.3

to be fulfilled. Due to limited time, we did not implement the CCM algorithm applied in MIC, as discussed in section 9.2.3.

9.4.7 FR08 - Provide encryption of payload

In section 9.2.3 we presented test results of packet encryption and decryption. We have successfully implemented an algorithm that encrypts a packet using a key shared between two devices. The opposing device is able to decrypt and read the payload, while interfering nodes are not. The level of confidentiality provided by the algorithm makes us certain we have accomplished implementing an encryption algorithm that keeps packet content secret.

9.5 Review of Non-functional requirements

In the following sections the implementation is evaluated against the non-functional requirements determined in chapter 5 on page 63.

9.5.1 NFR01 - Module-based application

One of the first tasks were to separate the program into modules, where every module performed a specific task according to the standard. Having one large program where everything was done sequentially instead of separating the tasks into multiple, smaller processes were something we wanted to achieve. The separation greatly helped us during our work and will benefit future development of this project. After the modularisation, we ended up with a large program split into smaller parts as can be seen in figures 8.3, 8.5, 8.6, 8.7 and 8.10. The figures display the packets in each of the respective devices, grouped by their task in the project.

By separating the program into clearly defined cases, we feel that NFR01 has been clearly achieved. It is easy to get an overview of what is happening where and the task of every function. Some of the modules can be further split up once the program grows and more functionality is implemented, but as of yet this has no meaning .

9.5.2 NFR02 - Failure resistant network (NFR02)

One of the major strengths of mesh networks are the ability to detect and repair a failing link or device. WirelessHART only provides partial mesh, where certain nodes are connected to multiple nodes. The network is still vulnerable to errors at the Gateway or Access Points as they act as sinks for traffic heading to the Gateway. Maintaining multiple Access Points will make sure that even in the event of a failing AP the network can reroute traffic to another sink, preventing total isolation. During the join process of a new node, we demand a minimum of two parenting nodes if possible, to reduce the risk of isolated nodes if their only connection should fail or leave the network. To discover errors in a network, nodes communicate at regular intervals. If nodes have not communicated for over 30 seconds,

a keep-alive packet is sent between the nodes to get status updates and synchronise clocks. If no response is returned, a fault is assumed and the incident is reported to the Network Manager. The Network Manager updates the routing tables, generates alternative routes for the affected routes, and transmits the changes to the concerning nodes. This is called a "messy exit", as the leaving device does not notify its neighbours of its departure. The opposite instant is when a node knows that it is about to leave, and sends a packet to its neighbours to notify about its departure. We handle both cases the same way, but the disadvantage of the messy departure is that some time might pass before the fault is discovered. During this period, many packets might have been lost due to the error.

9.5.3 NFR03/NFR04 - Multiple and separate Access Points

As discussed in section 8.6, we have separated the Access Points from the Gateway. The separation allowed us to have several Access Points, each having their own subnets. In our test network, we ran a total of nine devices and determined that two Access Points would be sufficient to provide enough coverage. Both Access Points were implemented with time synchronisation capabilities, although a small change could turn off the time provisioning.

9.5.4 NFR05 - General security

To demand that our program provides general security is a broad statement, and it can be discussed how general security it determined. For us, it was to ensure that two devices could safely communicate between them without any one else being able to read the exchanged packets. WirelessHART provides security at the Data Link Layer and the Network Layer, and by implementing the security functionality specified in these layers we concluded that security would be sufficient. FR07 and FR08 further elaborates on the specific security mechanisms and demonstrates how integrity and confidentiality is maintained during packet transmission.

9.6 Evaluation of development process

Evaluating the development process of a project cycle is difficult, especially when evaluating your own performance. The project got a slow start due to awaiting arrival of radio devices and multiple hardware failures on the equipment running the project, detailed in section 7.3 on page 79. Once these hurdles were dealt with, work was continued and the project were going well until the transfer limitation of JavaHidAPI [15] were discovered. Once again, the project had to be re-evaluated to determine the further route to take. We ended up emulating the project on a device acting as a WirelessHART node, which appeared to be quite successful.

As seen in table 9.3, several major changes were undergone during the course of the project. We have gone from working on proprietary nodes,

Initial	New	Reason for change
Atmel Raven boards	ABB Wimon 100	Inaccurate crystal clocks
ABB Wimon 100	Raspberry Pi	JavaHidAPI packet size limitation prevented sending proper WHART packets
Redhat Linux	Windows Xp	RavenUSB interfacing proved more reliable on Xp

Table 9.3: Project development

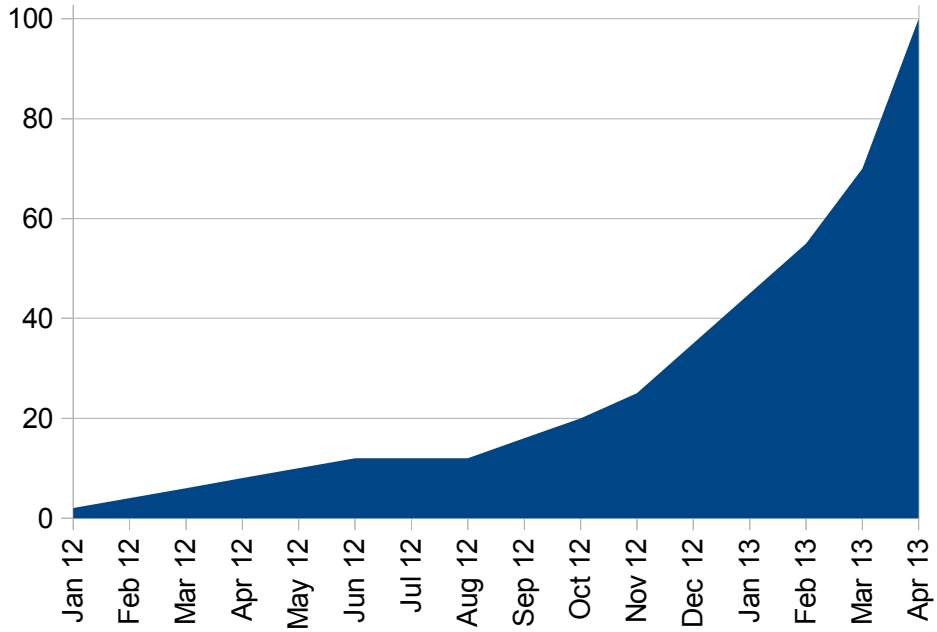

Figure 9.6: Workload spread over months

to simulating on Raspberry Pi which has changed the working situation to some extent. However, our main focus have been on the Gateway and Network Manager since the start. We feel that we have managed to highlight relevant issues on the area, and our proposed solutions meet the criteria specified in the project goals chapter at page 63.

The workload distribution over the months can be seen in figure 9.6. We got off at a slow start as was expected, which is how the master projects are scheduled at our program. The workload from January 2012 to August 2012 mostly consisted of studying the protocol and building a general foundation of knowledge. The workload arose steadily, but stopped during some of the major milestones as marked on the graph. Once the hurdles were overcome towards the end, the amount of work accelerated fast and the major parts of the thesis writing and implementation were done over the last months. Figure 9.7 displays the workload distribution between the tasks. As expected, studying the standard and writing the report have taken a large portion of the time as these are two time-consuming tasks. On

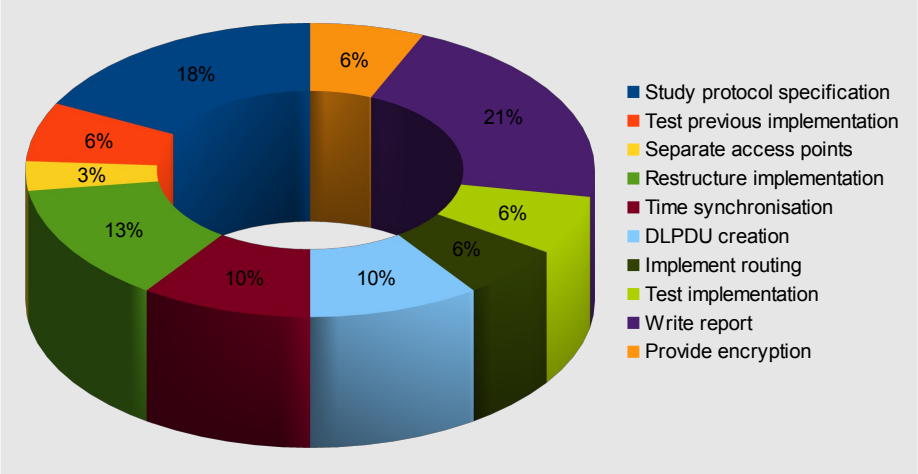

Figure 9.7: Work distribution between tasks

the implementation part, DLPDU creation and restructuring the Gateway have taken most time. Both tasks are very detailed and complicated, requiring a lot of time to implement every mechanism and value according to the standard.

9.7 Chapter Summary

The chapter has presented the reader with a test bed used to run the implementation. The implementation has been evaluated piece by piece, and test results has been evaluated against the requirements determined in chapter 5 to determine the level of accomplishment of each requirement. At the end, the development process and time distribution have been reviewed.

Chapter 10

Conclusion

In this report we have provided an overview of wireless network protocols and functionality, as well as a detailed description on how the WirelessHART protocol works. Our own work has been documented, and several key functionalities have been implemented. The implementation includes superframes, separate and multiple Access Points and a routing algorithm. We have also created the packet generation process where headers are applied as the packet passes through the different layers, before being successfully transmitted. The program has been restructured to correctly resemble the WirelessHART layered approach at the Gateway, where the layers communicate through interfaces.

We feel that the necessary restructuring of the program has been performed, as Asperheim et al stated in their thesis. The project has come to a stage where the foundation is according to the standard, but no further real work can be performed on proprietary nodes before the HCF tables listed in table C.1 have been acquired.

Chapter 11

Future Work

Our project has not covered everything needed for a full-fledged WirelessHART network. In the following sections we propose several key areas which are natural to continue working on.

11.1 Port to DresdenSam7s

RavenUSB and *DresdenSam7s* are the two prominent usb devices suitable for running the *15dot4-tools* project. The project is tailored for the *DresdenSam7s* devices according to Colin O'Flynn [1], which provides more memory and faster processing. It would be a natural and beneficial step to port the project to run on *DresdenSAM7s* devices, enabling the network to support larger amounts of traffic.

11.2 Develop libusb wrapper

In section 7.4 on page 84 we discussed the problems with using *HidAPI* for sending packets between the Network Manager and Access Points, as it only supports packet size up to 64 bytes. Only 32 bytes are available to the user, which is not enough for WirelessHART packets. Developing a *Libusb jni/jna*[1] wrapper for sending commands to *libusb* would circumvent the problem, and allow the user for sending packets of any size.

11.3 Channel blacklisting

WirelessHART has 16 channels that transmissions can channel hop on in order to provide more efficient communication between nodes. In the event channels are affected by consistent interferences, the channel can be put on a blacklist by the network administrator to prevent further use. We did not have the time to implement this function, but it is recommended under future work to provide better quality of service transmissions.

[1] Java Native Access

11.4 Channel offset

Calculation of the active channel is so far determined by
ActualChannel = (ChannelOffset + ASN) % NumChannels where *ChannelOffset* is set to zero to simplify the calculation. To further improve channel switching, implementing a function to distribute *ChannelOffset* between two communicating devices has to be implemented.

11.5 Session

According to the standard, a device has to request a session in order to communicate with another device. In our implementation, two devices communicate once the Network Manager has scheduled the communication. Future work would be to implement a method for the device to request a session from the Network Manager, which is responded to with a session the device can use for communication.

11.6 Quality of Service routing

Currently routing is only implemented on a least hop-count basis. Future quality of service routing could also provide routing based on traffic load on a link, device battery power, guaranteed delivery or signal level.

11.7 Source and Superframe routing

The current system only supports graph routing for transmitting a packet to its destination. The algorithm can be expanded to support additional routing algorithms by providing the devices with parts of the routing table and modifying the header fields in the packet.

11.8 HCF Enumeration tables

The time values implemented so far are specified in the WirelessHART standard [9]. More specific values are being referred to HCF enumeration tables currently not available to us. In order to successfully communicate with the proprietary nodes Wimon100, these values need to be implemented. A natural step would be to obtain the enumeration tables specified in table C.1 in appendix C on page 155.

Bibliography

[1] *15dot4-tools Project*. Nov. 2010. URL: http://www.newae.com/tiki-index.php?page=15dot4tools (visited on 15/04/2013).

[2] *ABB Norway*. ABB. URL: www.abb.no (visited on 10/11/2012).

[3] Kaja F.L. Skaar Anders Asperheim Rune V. Sjøen. 'Design and Implementation of a Rudimentary WirelessHART Network'. MA thesis. University of Oslo, Aug. 2012.

[4] *Atmel Corporation - Microcontrollers, 32-bit, and touch solutions*. Atmel. URL: www.atmel.com (visited on 08/04/2013).

[5] *Atmel Studio 6 - Supporting Two Architectures: AVR and ARM, with One Integrated Studio - Overview*. Atmel inc. URL: http://www.atmel.com/microsite/atmel_studio6/ (visited on 08/04/2013).

[6] Deji Chen, Mark Nixon and Aloysius Mok. *Wirelesshart: real-time mesh network for industrial automation*. Springer Publishing Company, Incorporated, 2010.

[7] Håvard Tegelsrud & Jørgen Frøysadal. 'WirelessHART Gjennomgang og implementering'. MA thesis. University of Oslo, May 2010.

[8] *HART Communication Foundation - History*. HART Communication Foundation. URL: http://www.hartcomm.org/hcf/aboutorg/aboutorg_history.html (visited on 08/04/2013).

[9] International Electrotechnical Commision (IEC). *Industrial communication networks - Fieldbus specifications - WirelessHART communication network and communication profile*. Tech. rep. International Electrotechnical Commision (IEC), 2009.

[10] *Institute of Automation, Automation Systems Group - Home*. URL: www.auto.tuwien.ac.at (visited on 24/11/2012).

[11] *ISA | The International Society of Automation*. Isa. URL: www.isa.org (visited on 08/04/2013).

[12] *javax.crypto (Java Platform SE 7)*. Oracle. URL: http://docs.oracle.com/javase/7/docs/api/javax/crypto/package-summary.html (visited on 09/04/2013).

[13] Zhao Jindong, Liang Zhenjun and Zhao Yaopei. 'ELHFR: A graph routing in industrial wireless mesh network'. In: *Information and Automation, 2009. ICIA'09. International Conference on*. IEEE. 2009, pp. 106–110.

[14] T. Lennvall, S. Svensson and F. Hekland. 'A comparison of WirelessHART and ZigBee for industrial applications'. In: *Factory Communication Systems, 2008. WFCS 2008. IEEE International Workshop on*. 2008, pp. 85–88. DOI: 10.1109/WFCS.2008.4638746.

[15] *signal11/hidapi. Github*. URL: https://github.com/signal11/hidapi (visited on 2013).

[16] *StatoilHydro uses HART Communication to Deliver Natural Gas from Under the Sea*. HART Communication Foundation. URL: http://www.hartcomm.org/protocol/applications/applications_success_statoil.html (visited on 08/04/2013).

[17] Andrew S. Tanenbaum. *Computer networks (4. ed.)* Prentice Hall, 2002, pp. I–XX, 1–891. ISBN: 978-0-13-038488-1.

[18] *The ABB Group - Automation and Power Technologies*. ABB. URL: www.abb.com (visited on 12/11/2012).

[19] *Wireshark . About*. Riverbed Technology. URL: http://www.wireshark.org/about.html (visited on 08/04/2013).

[20] *ZigBee Wireless Standard - Technology - Digi International*. Digi International Inc. URL: http://www.digi.com/technology/rf-articles/wireless-zigbee (visited on 08/04/2013).

Appendix A

Definitions

ACK - Acknowledgment

AP - Acess Point

API - Application Programming Interface

ASN - Absolute Slot Number

avgRSL - Average Received Signal

CCA - Clear Channel Assessment

CRC - Cyclic Redundancy Check

CSMA/CA - Carrier Sense Multiple Access Collision Avoidance

CSMA/CD - Carrier Sense Multiple Access Collision Detection

CTS - Clear To Send

DDL - Data Link Layer

DLPDU - Data Link Protocol Data Unit

DSSS - Direct Sequence Spread Spectrum

EUI - Equipment Unique Identifier

FHSS - Frequency Hopping Spread Spectrum

HART - Highway Addressable Remote Transducer

HCF - Hart Communication Foundation

HID - Human Interface Device

IDE - Integrated Development Environment

IEEE - Institute of Electrical and Electronics Engineers

ISA - International Society of Automation

JDK - Java Development Kit

JNI - Java Native Interface

JoinRspTimer - Join Response Timer

JRE - Java Runtime Environment

LLC - Logical Link Control

MAC - Medium Access Control

MIC - Message Integrity Check

NL - Network Layer

NM - Network Manager

NPDU - Network Protocol Data Unit

NTP - Network Time Protocol

OSI - Open Systems Interconnection

OS - Operating System

PAN - Personal Area Network

PDU - Protocol Data Unit

PIB - PAN Information Base

PPDU - Physical Protocol Data Unit

RECV - Receive

RNDIS - Remote Network Driver Interface Specification

RTS - Ready To Send

RX - Receive

SPDU - Security Protocol Data Unit

SP - Service Primitive

TDMA - Time Division Multiple Access

TTL - Time To Live

TX - Transmit

Appendix B

HART Commands

This appendix lists all implemented commands in our solution. A complete implementation of one of these commands can be found in appendix E.2

B.1 Universal (Commands 0-30 + 38, 48)

Command 20: Read Long Tag

This command request the receiving node to send its Unique ID. This is used in the join request. The Request is empty while the response contains an 8 byte body.

B.2 Additional Common Practice (Commands 512-767)

Command 961: Write Network Key

This is a Wireless Network Manager Command. It allows the Network Manager to write the network key on a Network Device. It is important that this command is rejected with Response Code 16 (Access Restricted) by the node if it is not sent from the Network Manager. The command is part of the join reply.

Byte	Format	Description
0-15	Unsigned-128	Key value
16-20	Unsigned-40	Execution time for command (ASN). 0 - execute immediately

Table B.1: Command 961 Request and Response Data Bytes

Code	Class	Description
0	Success	No Command-Specific Errors
5	Error	Too few Data Bytes Received
16	Error	Access Restricted
65	Error	Key change failed
66	Error	Invalid execution time

Table B.2: Command 961 Command-Specific Response Codes

B.3 WirelessHART (Commands 768-1023)

Provisioning

Command 962: Write Device Nickname Address

This is a Wireless Network Manager Command. It allows the Network Manager to set a Network Device's Nickname. This command shall be rejected with a response code 16 (Access Restricted) if the source is any device other than the Network Manager. It is used in Join Reply to tell the newly joined node its nickname.

Byte	Format	Description
0-1	Unsigned-16	Nickname

Table B.3: Command 962 Request and Response Data Bytes

Code	Class	Description
0	Success	No Command-Specific Errors
5	Error	Too few Data Bytes Received
16	Error	Access Restricted
65	Error	Invalid Nickname

Table B.4: Command 962 Command-Specific Response Codes

Managing Superframes and Links

Command 965: Write Superframe

If Superframe already is on the device, this will modify the data in the superframe specified. This command shall always be rejected with response code 16 if source is not the Network Manager.

Byte	Format	Description
0	Unsigned-8	SuperframeID
1-2	Unsigned-16	Number of slots in Superframe
3	Enum-8	Superframe Mode Flags
4	Unsigned-8	Reserved, shall be set to 0
5-9	Unsigned-40	Execution time of command (ASN). 0 - Execute immediately

Table B.5: Command 965 Request Data Bytes

Byte	Format	Description
0	Unsigned-8	SuperframeID
1-2	Unsigned-16	Number of slots in Superframe
3	Enum-8	Superframe Mode Flags
4	Unsigned-8	Number of Superframes remaining
5-9	Unsigned-40	Execution time of command (ASN)

Table B.6: Command 965 Response Data Bytes

Code	Class	Description
0	Success	No Command-Specific Errors
5	Error	Too few Data Bytes Received
16	Error	Access Restricted
65	Error	No more entries available
66	Error	Invalid number of slots
67	Error	Invalid Superframe mode

Table B.7: Command 965 Command-Specific Response Codes

Command 967: Write Link

This is a Wireless Network Manager command. It is used by the Network Manager to add a link assignment to a Network Device. The link is uniquely identified by the combination of the following key parameters: <Superframe ID, Slot Number, Neighbour Address>. Modification of existing links is not supported. A Network Manager shall delete an existing link and then write a new link to accomplish a modify operation. This command shall be rejected with a response code 16 (Access Restricted) if the source address is any device other than the Network Manager.

Byte	Format	Description
0	Unsigned-8	SuperframeID
1-2	Unsigned-16	Slot number in the Superframe for this link
3	Unsigned-8	channelOffset for this link
4-5	Unsigned-16	Nickname of the neighbour for this link (or 0xFFFF if broadcast link)
6	Bits-8	linkOptions (See HCF Enumeration Table 46. Link Option Flag Codes)
7	Enum-8	linkType (See HCF Enumeration Table 45. Link Type)

Table B.8: Command 967 Request Data Bytes

Byte	Format	Description
0	Unsigned-8	SuperframeID
1-2	Unsigned-16	Slot number in the Superframe for this link
3	Unsigned-8	channelOffset for this link
4-5	Unsigned-16	Nickname of the neighbour for this link (or 0xFFFF if broadcast link)
6	Bits-8	linkOptions (See HCF Enumeration Table 46. Link Option Flag Codes)
7	Enum-8	linkType (See HCF Enumeration Table 45. Link Type)
8-9	Unsigned-16	Number of link entries remaining

Table B.9: Command 967 Response Data Bytes

Code	Class	Description
0	Success	No Command-Specific Errors
5	Error	Too few Data Bytes Received
16	Error	Access Restricted
65	Error	No more links available
66	Error	Link already exists
67	Error	Unknown Superframe ID
68	Error	Invalid slot number
69	Error	Invalid link options (e.g., neither transmit nor receive set)
70	Error	Invalid channel offset
71	Error	Invalid link type
72	Error	No more neighbours available

Table B.10: Command 967 Command-Specific Response Codes

Managing Graph and Source Routes

Command 969: Write Graph/Neighbour Pair

This is a Wireless Network Layer Manager Command. It is used by the Network Manager to add a connection assignment for a Network

Device. This command shall be rejected with a response code of 16 (Access Restricted) if the source address is any device other than the Network Manager.

Byte	Format	Description
0-1	Unsigned-16	GraghID
2-3	Unsigned-16	Nickname of Neighbour

Table B.11: Command 969 Request Data Bytes

Byte	Format	Description
0-1	Unsigned-16	GraghID
2-3	Unsigned-16	Nickname of Neighbour
4	Unsigned-8	Number of Connections Remaining

Table B.12: Command 969 Response Data Bytes

Code	Class	Description
0	Success	No Command-Specific Errors
5	Error	Too few Data Bytes Received
16	Error	Access Restricted
65	Error	No more entries available
66	Error	Unknown Nickname

Table B.13: Command 969 Command-Specific Response Codes

Command 974: Write Route

This is a Wireless Network Manager Command. It allows the Network Manager to set or modify a route for a particular service. This command shall always be rejected with response code 16 (Access Restricted) if the source address is any other device than the Network Manager.

Byte	Format	Description
0	Unsigned-8	Route ID
1-2	Unsigned-16	Peer Nickname
3-4	Unsigned-16	Graph ID

Table B.14: Command 974 Request Data Bytes

Command 975: Delete Route

This is a Wireless Network Manager Command. It allows the Network Manager to delete a route that was previously written. It is deleted using the Route ID that was used to write the route information. This command shall be rejected with a response code 16 (Access Restricted) if the source address is any device other than the Network Manager.

Byte	Format	Description
0	Unsigned-8	Route ID
1-2	Unsigned-16	Peer Nickname
3-4	Unsigned-16	Graph ID
5	Unsigned-8	Number of Routes remaining.

Table B.15: Command 974 Response Data Bytes

Code	Class	Description
0	Success	No Command-Specific Errors
5	Error	Too few Data Bytes Received
16	Error	Access Restricted
65	Error	No more entries available
66	Error	Invalid Nickname
67	Error	Invalid Graph ID

Table B.16: Command 974 Command-Specific Response Codes

Byte	Format	Description
0	Unsigned-8	Route ID

Table B.17: Command 975 Request Data Bytes

Byte	Format	Description
0	Unsigned-8	Route ID
1	Unsigned-8	Number of Routes remaining.

Table B.18: Command 975 Response Data Bytes

Code	Class	Description
0	Success	No Command-Specific Errors
5	Error	Too few Data Bytes Received
16	Error	Access Restricted
65	Error	Invalid Route ID

Table B.19: Command 975 Command-Specific Response Codes

Command 976: Write Source-Route

This is a Wireless Network Manager Command. It is used by the Network Manager to write a source route. Broadcast addresses are not legal address values in this command and shall not be included in Source-Routes or the requests or responses for this command. This command shall be rejected with a response code of 16 (Access Restricted) if the source address is any device other than the Network Manager.

Byte	Format	Description
0	Unsigned-8	Route ID
1	Unsigned-8	Number of hops
2-3	Unsigned-16	Nickname of hop entry 0
4-...	Unsigned-16	Repeated for number of entries

Table B.20: Command 976 Request Data Bytes

Byte	Format	Description
0	Unsigned-8	Route ID
1	Unsigned-8	Number of Routes remaining.

Table B.21: Command 976 Response Data Bytes

Code	Class	Description
0	Success	No Command-Specific Errors
5	Error	Too few Data Bytes Received
8	Warning	Broadcast addresses deleted in response
16	Error	Access Restricted
65	Error	No more entries available
66	Error	Invalid Route ID
67	Error	Invalid Nickname
68	Error	Invalid number of hops

Table B.22: Command 976 Command-Specific Response Codes

Security

Command 961: Write Network Key

This is a Wireless Network Manager Command. It allows the Network Manager to write the network key on a Network Device. This command shall be rejected with response code 16 (Access Restricted) if the source address is any device other than the Network Manager. The command is used in the Join Reply.

Byte	Format	Description
0-15	Unsigned-128	Key value
16-20	Unsigned-40	Execution time for command (ASN). 0 - execute immediately
3	Enum-8	Superframe Mode Flags
4	Unsigned-8	Number of Superframes remaining
5-9	Unsigned-40	Execution time of command (ASN)

Table B.23: Command 961 Request and Response Data Bytes

Code	Class	Description
0	Success	No Command-Specific Errors
5	Error	Too few Data Bytes Received
16	Error	Access Restricted
65	Error	Key change failed
66	Error	Invalid execution time

Table B.24: Command 961 Command-Specific Response Codes

Command 963: Write Session

This is a Wireless Network Manager Command. This command allows the Network Manager to write the session parameters required to establish a session between the device the message is addressed to and the peer device contained in the request. This command shall be rejected with a response code of 16 (Access Restricted) if the source address is any device other than the Network Manager. Used in Join Reply to create the session for initiating the node.

Byte	Format	Description
0	Enum-8	Session type. (See HCF Enumeration Table 48. Session Type Code)
1-2	Unsigned-16	Nickname of peer device
3-4	Unsigned-16	Peer expanded device type code
5-7	Unsigned-24	Peer device Id
8-11	Unsigned-32	Peer Nounce counter value
12-27	Unsigned-128	Key value
28	Unsigned-8	Reserved shall be set to 0
29-31	Unsigned-40	Execution time of command (ASN)

Table B.25: Command 963 Request Data Bytes

Byte	Format	Description
0	Enum-8	Session type. (See HCF Enumeration Table 48. Session Type Code)
1-2	Unsigned-16	Nickname of peer device
3-4	Unsigned-16	Peer expanded device type code
5-7	Unsigned-24	Peer device Id
8-11	Unsigned-32	Peer Nounce counter value
12-27	Unsigned-128	Key value
28	Unsigned-8	Number of sessions remaining
29-31	Unsigned-40	Execution time of command (ASN)

Table B.26: Command 963 Response Data Bytes

Code	Class	Description
0	Success	No Command-Specific Errors
5	Error	Too few Data Bytes Received
16	Error	Access Restricted
65	Error	No more entries available
66	Error	Session type invalid

Table B.27: Command 963 Command-Specific Response Codes

Bandwidth Management

No commands implemented!

Device Management

Command 960: Disconnect Device

This is a Wireless Network Manager Command. It allows the Network Manager to force a device off the network, clear all its network information and rejoin the network. This command shall be rejected with response code 16 (Access Restricted) if the source address is any device other than the Network Manager.

Byte	Format	Description
0	Unsigned-8	Reason. (See HCF Enumeration Table 50. Disconnect Cause Codes)

Table B.28: Command 960 Request and Response Data Bytes

Code	Class	Description
0	Success	No Command-Specific Errors
16	Error	Access Restricted

Table B.29: Command 960 Command-Specific Response Codes

Network Maintenance

Command 795: Write Timer Interval

This is a Wireless Data Link Command. This command allows the Network Manager to set an interval for a timer on a Network Device.

Byte	Format	Description
0	Enum-8	Timer Type. (See HCF Enumeration Table 43. Wireless Timer Code)
1-2	Unsigned-32	Timer Interval (in ms)

Table B.30: Command 795 Request and Response Data Bytes

Code	Class	Description
0	Success	No Command-Specific Errors
3	Error	Passed Parameter Too Large
4	Error	Passed Parameter Too Small
5	Error	Too Few Data Bytes Received
6	Error	Device-Specific Command Error
7	Error	In Write Protected Mode
8	Warning	Set To Nearest Possible Value
16	Error	Access Restricted (i.e., this command shall be accepted only from the Network Manager)
32	Error	Busy (A DR Could Not Be Started)
33	Error	DR Initiated
34	Error	DR Running
35	Error	DR Dead
36	Error	DR Conflict
65	Error	Invalid Timer Type
66	Error	Invalid Timer Interval

Table B.31: Command 795 Command-Specific Response Codes

Coexistence

No commands implemented!

Network Health Reporting and Status

Command 787: Report Neighbour Signal Levels

This command is periodically published to the Network Manager (see Command 795 and HCF Enumeration Table 43) indicating discovered (but not linked) neighbours. The command response is transmitted ad "Process Data" level priority. Command 780 reports linked neighbours. The neighbour table is treated as a list with entries being added as neighbours are detected. Neighbours with links to device are at the beginning of the list and neighbours without links at the end. Used in Join Request.

Byte	Format	Description
0	Unsigned-8	Neighbour table index
1	Unsigned-8	Number of neighbour entries to read

Table B.32: Command 787 Request Data Bytes

Byte	Format	Description
0	Unsigned-8	Neighbour table index
1	Unsigned-8	Number of neighbour entries to read
2	Unsigned-8	Total number of neighbours
3-4	Unsigned-16	Nickname of neighbour
5	Signed-8	RSL of neighbour in dB
6-8		Repeats (as needed) based on response byte 1

Table B.33: Command 787 Response Data Bytes

Code	Class	Description
0	Success	No Command-specific Errors
5	Error	Too Few Data Bytes Received
8	Warning	Set To Nearest Possible Value

Table B.34: Command 787 Command-Specific Response Codes

Gateway Commands

No commands implemented!

Appendix C

HCF Tables

Protocol	Revision	Price
HART Field Communication Protocol Specifications	7.4 (HCF_KIT-13)	$975.00
HART Field Communication Protocol Test Specifications	7.4 (HCF_KIT-210)	$500.00
DDL Specifications	(HCF_KIT-215)	$250.00
HART Field Communications Protocol, A Technical Overview	(HCF_LIT-020)	$60.00
The HART Data Link Layer, A Requirement Analysis	(HCF_LIT-067)	$50.00
Application Note: Information for HART Modem Designers	(HCF_LIT-073)	$50.00
HART Server	(HCF_KIT-175)	$500.00

Table C.1: Required HCF_tables

Appendix D

Runtime problems

D.0.1 Access Point problem during runtime

```
Jan 30, 2013 1:17:37 PM no.uio.ifi.wirelesshart.ui.
NetManShell <init>
INFO: Starting shell..
Jan 30, 2013 1:17:37 PM
no.uio.ifi.wirelesshart.manager.NetworkManager <init>
INFO: Instantiating Network Manager
Jan 30, 2013 1:17:37 PM
no.uio.ifi.wirelesshart.accesspoint.pal.
HIDRawUSBpalgetHIDDevice
INFO: Failed to connect to Access Point.
Trying to open device
Failed to open device.
Jan 30, 2013 1:17:37 PM
no.uio.ifi.wirelesshart.gateway.VirtualGateway <init>
INFO: Started Network Manager...
Jan 30, 2013 1:17:37 PM
no.uio.ifi.wirelesshart.gateway.VirtualGateway <init>
INFO: Started Access Points...
Jan 30, 2013 1:17:37 PM
no.uio.ifi.wirelesshart.gateway.VirtualGateway run
INFO: Started Virtual Gateway...
```

D.0.2 Buffer problems moving to Windows

```

jan 30, 2013 1:47:57 PM no.uio.ifi.wirelesshart.ui.
   ↪ NetManShell
<init>
INFO: Starting shell..
jan 30, 2013 1:47:57 PM
no.uio.ifi.wirelesshart.manager.NetworkManager <init>
```

```
INFO: Instantiating Network Manager
jan 30, 2013 1:47:58 PM
no.uio.ifi.wirelesshart.accesspoint.pal.
HIDRawUSBpal getHIDDevice
INFO: Found Atmel device "ATMEL 8226"
jan 30, 2013 1:47:58 PM no.uio.ifi.wirelesshart.
   ↪ accesspoint.pal.
HIDRawUSBpalgetHIDDevice
INFO: Connecting to: \textbackslash \textbackslash
? \textbackslash
hid\#vid\_03eb\&pid\_2022\&mi\_02\#7\&28cec948
   ↪ \&0\&0000\#{4d1e55
b2-f16f-11cf-88cb-001111000030} jan 30, 2013 1:47:58
   ↪ PM
no.uio.ifi.wirelesshart.accesspoint.pal.HIDRawUSBpal
   ↪ <init>
INFO: Successfully connected to device.
jan 30, 2013 1:47:58 PM
no.uio.ifi.wirelesshart.gateway.VirtualGateway <init>
INFO: Started Network Manager...
jan 30, 2013 1:47:58 PM
no.uio.ifi.wirelesshart.gateway.VirtualGateway <init>
INFO: Started Access Points...
jan 30, 2013 1:47:58 PM
no.uio.ifi.wirelesshart.gateway.VirtualGateway run
INFO: Started Virtual Gateway...
java.io.IOException: The supplied user buffer is not
   ↪ valid for
the requested operation.
at com.codeminders.hidapi.HIDDevice.write(Native
   ↪ Method) at
no.uio.ifi.wirelesshart.accesspoint.pal.HIDRawUSBpal.
   ↪ send(HIDRaw
USBpal.java:103) at
no.uio.ifi.wirelesshart.accesspoint.dll.HIDRawUSBdll.
   ↪ send(HIDRaw
USBdll.java:17) at
no.uio.ifi.wirelesshart.accesspoint.net.HIDRawUSBnet.
   ↪ send(HIDRaw
USBnet.java:15) at
no.uio.ifi.wirelesshart.accesspoint.tl.HIDRawUSBtl.
   ↪ send(HIDRawUS
Btl.java:15) at
no.uio.ifi.wirelesshart.accesspoint.session.
   ↪ HIDRawUSBses.send(HI
DRawUSBses.java:15) at
no.uio.ifi.wirelesshart.accesspoint.
   ↪ HIDRawUSBAccessPoint.send(HI
```

```
43 DRawUSBAccessPoint.java:195) at
44 no.uio.ifi.wirelesshart.accesspoint.
   ↪ HIDRawUSBAccessPoint.run(HID
45 RawUSBAccessPoint.java:120)
46 ASN: 5501567370 ASN2: 5502145209 wait: 552630 ms: 0
   ↪ ns: 552630
```

Appendix E

Code

In this appendix there are specific code from the implementation that is being refered to in the thesis.

E.1 DLPDU

The Data Link Protocol Data Unit. This is one of the most advanced DPUs in the implementation.

```
1
2  package no.uio.ifi.wirelesshart.devicenetworkmanagement.pdu;
3
4  /**
5   * TODO Add Network key (bit 3 in DLPDUSpecifier)
6   *
7   * @author Magnus
8   * @date 27. feb. 2013
9   *
10  */
11 public class DLPDU {
12     public enum PayloadType {
13
14         ACK, ADVERTISE, KEEP_ALIVE, DISCONNECT, DATA;
15
16         static PayloadType getPayloadType(byte b) {
17             if((b & 0b00000111) == 0b00000111) return DATA ;
18
19             if((b & 0b00000011) == 0b00000011) return
                   ↪ DISCONNECT ;
20
21             if((b & 0b00000010) == 0b00000010) return
                   ↪ KEEP_ALIVE ;
22
23             if((b & 0b00000001) == 0b00000001) return
                   ↪ ADVERTISE ;
24
```

```
25              if((b & 0b00000000) == 0b00000000) return ACK ;
26
27              return null ;
28          }
29
30      }
31
32      public enum Priority {
33
34          COMMAND, PROCESS_DATA, NORMAL, ALARM;
35
36          static Priority getPriority(byte b) {
37              if((b & 0b00110000) == 0b00110000) return COMMAND
                    ↪ ;
38
39              if((b & 0b00100000) == 0b00100000) return
                    ↪ PROCESS_DATA ;
40
41              if((b & 0b00010000) == 0b00010000) return NORMAL ;
42
43              if((b & 0b00000000) == 0b00000000) return ALARM ;
44
45              return null ;
46          }
47      }
48
49      final static byte startByte = (byte) 0x41 ;
50      byte addressSpecifier  = (byte) 0x88 ;
51      byte sequenceNumber ;
52
53      byte[] networkID ;
54      byte[] srcAddr ;
55      byte[] dstAddr ;
56
57      byte dlpduSpecifier ;
58
59      Priority priority ;
60
61      PayloadType payloadType ;
62      byte[] payload ;
63
64      byte[] mic = new byte[4] ;
65      byte[] crc = new byte[2] ;
66
67      int sizeOfHeader = 6 ;
68      int sizeOfPayload ;
69      int sizeOfTail = 6 ;
70
```

```
71      /**
72       * Constructor for the DLPDU when recieveing a bytearray
73       *
74       * @param data bytearray setup as a dlpdu.
75       */
76      public DLPDU(byte[] data) {
77          ByteArrayInputStream bais = new ByteArrayInputStream(
                ↪ data) ;
78          DataInputStream dis = new DataInputStream(bais) ;
79
80          try{
81              dis.readByte() ;
82              addressSpecifier = dis.readByte() ;
83              sequenceNumber = dis.readByte() ;
84
85              networkID = new byte[2] ;
86              dis.read(networkID) ;
87
88              if((addressSpecifier & (1 << 2)) == 1 << 2) {
89                  dstAddr = new byte[8] ;
90                  dis.read(dstAddr) ;
91                  sizeOfHeader += 8 ;
92              }else {
93                  dstAddr = new byte[2] ;
94                  dis.read(dstAddr) ;
95                  sizeOfHeader += 2 ;
96              }
97
98              if((addressSpecifier & (1 << 6)) == 1 << 6) {
99                  srcAddr = new byte[8] ;
100                 dis.read(srcAddr) ;
101                 sizeOfHeader += 8 ;
102             }else {
103                 srcAddr = new byte[2] ;
104                 dis.read(srcAddr) ;
105                 sizeOfHeader += 2 ;
106             }
107
108             dlpduSpecifier = dis.readByte() ;
109
110             payloadType = PayloadType.getPayloadType(
                    ↪ dlpduSpecifier) ;
111             priority = Priority.getPriority(dlpduSpecifier) ;
112
113             sizeOfPayload = data.length - sizeOfHeader -
                    ↪ sizeOfTail ;
114             payload = new byte[sizeOfPayload] ;
115
```

```
116             dis.read(payload) ;
117
118             mic = new byte[4] ;
119             dis.read(mic) ;
120
121             crc = new byte[2] ;
122             dis.read(crc) ;
123
124
125         } catch(IOException e) {
126             e.printStackTrace() ;
127         }
128     }
129
130     /**
131      * Constructor when generating a DLPDU in the Network
            ↪ Manager
132      *
133      * @param sequenceNumber is the least significant byte in
            ↪ the ASN
134      * @param networkID 2 byte networkID
135      * @param dstAddr Next hop dst addr (2 or 8 bytes)
136      * @param srcAddr This node's addr (2 or 8 bytes)
137      * @param priority The priority of the DLPDU. Command is
            ↪ highest priority, Alarm is lowest.
138      * @param payloadType The type of the payload.
139      * @param payload A byte array of the payload, depending
            ↪ on the payloadType
140      */
141     public DLPDU(byte sequenceNumber, byte[] networkID, byte[]
        ↪  dstAddr,
142             byte[] srcAddr, Priority priority, PayloadType
                ↪ payloadType, byte[] payload) {
143         this.sequenceNumber = sequenceNumber ;
144         this.networkID = networkID ;
145         this.dstAddr = dstAddr ;
146         this.srcAddr = srcAddr ;
147
148         if(dstAddr.length == 8) addressSpecifier = (byte) (
                ↪ addressSpecifier |= 1 << 2) ;
149         if(srcAddr.length == 8) addressSpecifier = (byte) (
                ↪ addressSpecifier |= 1 << 6) ;
150
151         this.priority = priority ;
152
153         this.payloadType = payloadType ;
154         this.payload = payload ;
155
```

```
156             generateDLPDUSpecifier() ;
157
158             generateMIC() ;
159             generateCRC() ;
160
161             sizeOfPayload = payload.length ;
162             sizeOfHeader += dstAddr.length + srcAddr.length ;
163         }
164
165         /**
166          * Generates a MIC for the PDU.
167          * This must be called after all variables have been set!
168          *
169          * @author Magnus
170          * @date 5. mars 2013
171          */
172         public void generateMIC() {
173             mic = MIC.generateMIC(payload) ;
174         }
175
176         public boolean confirmMIC() {
177             return MIC.confirmMIC(payload) ;
178         }
179
180         /**
181          * Generates a CRC for the PDU.
182          * This must be called after all variables have been set!
183          *
184          * @author Magnus
185          * @date 5. mars 2013
186          */
187         public void generateCRC() {
188             crc =  CRC.generateCRC(payload) ;
189         }
190
191         public boolean confirmCRC() {
192             return CRC.confirmCRC(payload) ;
193         }
194
195         /**
196          * This method creates the DLPDUSpesifier byte dependent
                ↪ on the priority,
197          * network key and packet type
198          *
199          * @author Magnus
200          * @date 5. mars 2013
201          */
202         private void generateDLPDUSpecifier() {
```

```
203             switch(priority) {
204             case ALARM:
205                 break;
206             case COMMAND:
207                 dlpduSpecifier = (byte) (dlpduSpecifier |= 1 << 5)
                        ↪  ;
208                 dlpduSpecifier = (byte) (dlpduSpecifier |= 1 << 4)
                        ↪  ;
209                 break;
210             case NORMAL:
211                 dlpduSpecifier = (byte) (dlpduSpecifier |= 1 << 4)
                        ↪  ;
212                 break;
213             case PROCESS_DATA:
214                 dlpduSpecifier = (byte) (dlpduSpecifier |= 1 << 5)
                        ↪  ;
215                 break;
216             }
217
218             switch(payloadType) {
219             case ACK:
220                 break;
221             case ADVERTISE:
222                 dlpduSpecifier = (byte) (dlpduSpecifier |= 1 << 0)
                        ↪  ;
223                 break;
224             case DATA:
225                 dlpduSpecifier = (byte) (dlpduSpecifier |= 1 << 0)
                        ↪  ;
226                 dlpduSpecifier = (byte) (dlpduSpecifier |= 1 << 1)
                        ↪  ;
227                 dlpduSpecifier = (byte) (dlpduSpecifier |= 1 << 2)
                        ↪  ;
228                 break;
229             case DISCONNECT:
230                 dlpduSpecifier = (byte) (dlpduSpecifier |= 1 << 0)
                        ↪  ;
231                 dlpduSpecifier = (byte) (dlpduSpecifier |= 1 << 1)
                        ↪  ;
232                 break;
233             case KEEP_ALIVE:
234                 dlpduSpecifier = (byte) (dlpduSpecifier |= 1 << 1)
                        ↪  ;
235                 break;
236             }
237         }
238
239         /**
```

```
240         *
241         * @return The object as a byte array for sending over
                ↪ socket
242         *
243         * @author Magnus
244         * @date 5. mars 2013
245         */
246        public byte[] toByteArray() {
247
248            ByteArrayOutputStream baos = new ByteArrayOutputStream
                   ↪ () ;
249            DataOutputStream dos = new DataOutputStream(baos) ;
250
251            try{
252                dos.write(startByte) ;
253                dos.write(addressSpecifier) ;
254                dos.write(sequenceNumber) ;
255                dos.write(networkID) ;
256                dos.write(dstAddr) ;
257                dos.write(srcAddr) ;
258                dos.write(dlpduSpecifier) ;
259                dos.write(payload) ;
260                dos.write(mic) ;
261                dos.write(crc) ;
262
263            } catch(IOException e) {
264                e.printStackTrace() ;
265            }
266
267            return baos.toByteArray() ;
268        }
269    }
270
271    }
```

Listing E.1: The DLPDU

E.2 Command 787: Report Neighbor Signal Levels

The implementation of command 787: Report Neighbor Signal Levels. All commands extend the Command class seen in listing 8.8.

```
1
2  package no.uio.ifi.wirelesshart.applicationlayer.commands;
3
4  /**
5   * Command 787 Report Neighbor Signal Levels
6   *
7   * size of command should be:
8   * REQUEST: 2 bytes
9   * RESPONSE: 3 + (3*numberOfNeighborEntriesToRead)
10  *
11  * @author Magnus
12  *
13  */
14 public class ReportNeighborSignalLevels extends Command {
15
16     //GENERAL DATA BYTES
17     byte neighborTableIndex ; //byte 0
18     byte numberOfNeighborEntriesToRead = 0 ; //byte 1
19
20     //RESPONSE DATA BYTES
21     byte totalNumerOfNeighbors ; //byte 2
22
23     ArrayList<Node> nodes = new ArrayList<Node>() ;
24
25     byte[] shortByteArray = new byte[2] ;
26
27     /**
28      * Build from bytes constructor
29      *
30      * @param data raw byte array of the structure
31      */
32     public ReportNeighborSignalLevels(byte[] data) {
33         super(CommandName.REPORT_NEIGHBOR_SIGNAL_LEVELS) ;
34
35         ByteArrayInputStream bais = new ByteArrayInputStream(
               ↪ data) ;
36         DataInputStream dis = new DataInputStream(bais) ;
37
38         try {
39             neighborTableIndex = dis.readByte() ;
40             numberOfNeighborEntriesToRead = dis.readByte() ;
41             length = 2 ;
42
```

```
43              if(data.length > 2) {
44                  totalNumerOfNeighbors = dis.readByte() ;
45                  length++ ;
46
47                  for(int i = 0; i <
                        ↪ numberOfNeighborEntriesToRead; i++) {
48                      dis.read(shortByteArray) ;
49                      byte rsl = dis.readByte() ;
50
51                      nodes.add(new Node(HelpMethods.toShort(
                            ↪ shortByteArray), rsl)) ;
52                      length += 3 ;
53                  }
54
55              }
56
57              dis.close() ;
58
59          }catch(IOException e) {
60              e.printStackTrace() ;
61          }
62      }
63
64      /**
65       * Request constructor
66       *
67       * @param neighborTableIndex table index of neighbors
68       */
69      public ReportNeighborSignalLevels(byte neighborTableIndex)
            ↪  {
70          super(CommandName.REPORT_NEIGHBOR_SIGNAL_LEVELS) ;
71
72          this.neighborTableIndex = neighborTableIndex ;
73          totalNumerOfNeighbors = -1 ;
74          length = 1 ;
75      }
76
77      /**
78       * Response constructor
79       *
80       * @param neighborTableIndex table index of neighbors
81       * @param totalNumberOfNeighbors total number of neighbors
             ↪  of the node
82       */
83      public ReportNeighborSignalLevels(byte neighborTableIndex,
            ↪  byte totalNumberOfNeighbors) {
84          super(CommandName.REPORT_NEIGHBOR_SIGNAL_LEVELS) ;
85
```

```
86          this.neighborTableIndex = neighborTableIndex ;
87          this.totalNumerOfNeighbors = totalNumberOfNeighbors ;
88          length = 3 ;
89      }
90
91
92
93      @Override
94      public byte[] toByteArray() {
95          ByteArrayOutputStream baos = new ByteArrayOutputStream
               ↪ ();
96          DataOutputStream dos = new DataOutputStream(baos);
97
98          try {
99              dos.write(HelpMethods.toByteArray(getCommandNumber
                   ↪ ())) ;
100
101             if(commandName.getSize() != -1)
102                 dos.write(commandName.getSize()) ;
103             else
104                 dos.write(length) ;
105
106             dos.write(neighborTableIndex) ;
107             dos.write(numberOfNeighborEntriesToRead) ;
108
109             if(totalNumerOfNeighbors != -1) {
110                 dos.write(totalNumerOfNeighbors) ;
111
112                 for(Node n : nodes) {
113                     dos.write(HelpMethods.toByteArray(n.
                           ↪ neighborNickname)) ;
114                     dos.write(n.rsl) ;
115                 }
116             }
117
118         } catch(IOException e) {
119             e.printStackTrace() ;
120         }
121
122         return baos.toByteArray() ;
123     }
124
125     @Override
126     public ResponseCode executeCommand() {
127         // TODO Not implemented as of 15.apr. 2013.
128         // REMEMBER TO UPDATE APPENDIX WHEN DONE!
129         return null;
130     }
```

```
131
132     @Override
133     public byte[] generateResponse() {
134         return toByteArray() ;
135     }
136
137     /* ********************
138      * Getters and Setters *
139      * ******************** */
140
141     /**
142      * @param nickname nickname of the neighbor
143      * @param rsl the nodes average Received Signal Level
144      *
145      * @author Magnus
146      * @date 15. apr. 2013
147      */
148     public void addNeighbor(short nickname, byte rsl) {
149         nodes.add(new Node(nickname, rsl)) ;
150         length += 3 ;
151         numberOfNeighborEntriesToRead++ ;
152     }
153
154     /**
155      * @return the Table index of the neighbor
156      *
157      * @author Magnus
158      * @date 15. apr. 2013
159      */
160     public byte getNeighborTableIndex() {
161         return neighborTableIndex;
162     }
163
164     /**
165      * @return number of Neighbors in this report
166      *
167      * @author Magnus
168      * @date 15. apr. 2013
169      */
170     public byte getNumberOfNeighborEntriesToRead() {
171         return numberOfNeighborEntriesToRead;
172     }
173
174     /**
175      * @return The total number of neighbors at the node (
         ↪ response only)
176      *
177      * @author Magnus
```

```
178         * @date 15. apr. 2013
179         */
180        public byte getTotalNumerOfNeighbors() {
181            return totalNumerOfNeighbors;
182        }
183
184
185        /**
186         * @return a Hashmap of the neighbors on the node (
               ↪ response only)
187         *
188         * @author Magnus
189         * @date 15. apr. 2013
190         */
191        public Map<Short, Byte> getNeighbors() {
192            Map<Short, Byte> tmp = new HashMap<Short, Byte>() ;
193
194            for(Node n : nodes) {
195                tmp.put(n.neighborNickname, n.rsl) ;
196            }
197
198            return tmp ;
199        }
200
201        private class Node {
202            short neighborNickname ;
203            byte rsl ;
204
205            public Node(short neighborNickname, byte rsl) {
206                this.neighborNickname = neighborNickname ;
207                this.rsl = rsl ;
208            }
209
210        }
211
212    }
```

Listing E.2: Command 787: Report Neighbor Signal Levels

E.3 CCM implementation

In listing E.3 we have provided the code currently implemented for our CCM algorithm. The code is not yet tested, but should provide the foundation needed for finishing the algorithm.

```
1  package no.uio.ifi.wirelesshart.securitylayer;
2
3  import java.io.ByteArrayOutputStream;
```

```
4
5  import org.bouncycastle.crypto.BlockCipher;
6  import org.bouncycastle.crypto.CipherParameters;
7  import org.bouncycastle.crypto.DataLengthException;
8  import org.bouncycastle.crypto.InvalidCipherTextException;
9  import org.bouncycastle.crypto.Mac;
10 import org.bouncycastle.crypto.macs.CBCBlockCipherMac;
11 import org.bouncycastle.crypto.modes.AEADBlockCipher;
12 import org.bouncycastle.crypto.modes.SICBlockCipher;
13 import org.bouncycastle.crypto.params.AEADParameters;
14 import org.bouncycastle.crypto.params.ParametersWithIV;
15 import org.bouncycastle.util.Arrays;
16
17 public class L4_testCCM implements AEADBlockCipher {
18     private BlockCipher           cipher;
19     private int                   blockSize;
20     private boolean               forEncryption;
21     private byte[]                nounce;
22     private byte[]                associatedText;
23     private int                   macSize;
24     private CipherParameters      keyParam;
25     private byte[]                macBlock;
26     private ByteArrayOutputStream data = new
          ↪ ByteArrayOutputStream();
27
28
29     public L4_testCCM(BlockCipher c){
30         this.cipher = c;
31         this.blockSize = c.getBlockSize();
32         this.macBlock = new byte[blockSize];
33
34         if(blockSize != 16){
35             throw new IllegalArgumentException("cipher
                  ↪ required with a block size of 16.");
36         }
37     }
38
39
40
41     public int doFinal(byte[] out, int outOff) throws
          ↪ IllegalStateException,
42             InvalidCipherTextException {
43         byte[] text = data.toByteArray();
44         byte[] enc = processPacket(text, 0, text.length);
45
46         System.arraycopy(enc, 0, out, outOff, enc.length);
47         reset();
48         return enc.length;
```

```
49      }
50
51      public String getAlgorithmName() {
52          return cipher.getAlgorithmName() + "/CCM";
53      }
54
55      public byte[] getMac() {
56          byte[] mac = new byte[macSize];
57          System.arraycopy(macBlock, 0, mac, 0, mac.length);
58          return mac;
59      }
60
61
62      public int getOutputSize(int len) {
63          if(forEncryption)
64              return data.size() + len + macSize;
65          else
66              return data.size() + len - macSize;
67      }
68
69      public BlockCipher getUnderlyingCipher() {
70          return cipher;
71      }
72
73      public int getUpdateOutputSize(int len) {
74          return 0;
75      }
76
77
78      public void init(boolean forEncryption, CipherParameters
           ↪ params)
79              throws IllegalArgumentException {
80          this.forEncryption = forEncryption;
81
82          if(params instanceof AEADParameters){
83              AEADParameters param = (AEADParameters) params;
84
85              nounce = param.getNonce();
86              associatedText = param.getAssociatedText();
87              macSize = param.getMacSize() / 8;
88              keyParam = param.getKey();
89          }
90          else if(params instanceof ParametersWithIV){
91              ParametersWithIV param = (ParametersWithIV) params
                   ↪ ;
92
93              nounce = param.getIV();
94              associatedText = null;
```

```
95              macSize = macBlock.length / 2;
96              keyParam = param.getParameters();
97          }
98          else{
99              throw new IllegalArgumentException("invalid
                    ↪ parameters passed to CCM");
100         }
101
102     }
103
104     public int processByte(byte in, byte[] out, int outOff)
105             throws DataLengthException {
106         data.write(in);
107         return 0;
108     }
109
110     public int processBytes(byte[] in, int inOff, int inLen,
            ↪ byte[] out,
111             int outOff) throws DataLengthException {
112
113         data.write(in, inOff, inLen);
114         return 0;
115     }
116
117     public void reset() {
118         cipher.reset();
119         data.reset();
120     }
121
122     private boolean hasAssociatedText(){
123         return associatedText != null & associatedText.length
                ↪ != 0;
124     }
125
126     private int calculateMac(byte[] data, int dataOff, int
            ↪ dataLen, byte[] macBlock){
127         Mac cMac = new CBCBlockCipherMac(cipher, macSize * 8);
128         cMac.init(keyParam);
129
130         byte[] b0 = new byte[16];
131
132         if(hasAssociatedText())
133             b0[0] |= 0x40;
134
135         b0[0] |= (((cMac.getMacSize() - 2) / 2) & 0x7) << 3;
136         b0[0] |= ((15 - nounce.length) - 1) & 0x7;
137
138         System.arraycopy(nounce, 0, b0, 1, nounce.length);
```

```
139
140         int q = dataLen;
141         int count = 1;
142         while(q > 0){
143             b0[b0.length - count] = (byte) (q & 0xff);
144             q >>>= 8;
145             count++;
146         }
147
148         cMac.update(b0, 0, b0.length);
149
150         /*
151          * Process associated text
152          */
153
154         if(hasAssociatedText()){
155             int extra;
156
157             if(associatedText.length < ((1 << 16) - (1 << 8)))
                    ↪ {
158                 cMac.update((byte) (associatedText.length >>
                        ↪ 8));
159                 cMac.update((byte) associatedText.length);
160                 extra = 2;
161             }
162             else{
163                 cMac.update((byte) 0xff);
164                 cMac.update((byte) 0xfe);
165                 cMac.update((byte) (associatedText.length >>
                        ↪ 24));
166                 cMac.update((byte) (associatedText.length >>
                        ↪ 16));
167                 cMac.update((byte) (associatedText.length >>
                        ↪ 8));
168                 cMac.update((byte) associatedText.length);
169
170                 extra = 6;
171             }
172             cMac.update(associatedText, 0, associatedText.
                    ↪ length);
173
174             extra = (extra + associatedText.length) % 16;
175             if(extra != 0){
176                 for(int i = 0; i != 16 - extra; i++)
177                     cMac.update((byte) 0x00);
178             }
179         }
180         /*
```

```
181             * add the text
182             */
183            cMac.update(data, dataOff, dataLen);
184            return cMac.doFinal(macBlock,  0);
185        }
186
187        public byte[] processPacket(byte[] in, int inOff, int
               ↪ inLen)
188            throws IllegalStateException,
                   ↪ InvalidCipherTextException{
189
190            if(keyParam == null)
191                throw new IllegalStateException("CCM cipher
                       ↪ uninitialized");
192
193            BlockCipher ctrCipher = new SICBlockCipher(cipher);
194            byte[] iv = new byte[blockSize];
195            byte[] out;
196
197            iv[0] = (byte) (((15 - nounce.length) -1) &0x7);
198            System.arraycopy(nounce, 0, iv, 1, nounce.length);
199            ctrCipher.init(forEncryption, new ParametersWithIV(
                   ↪ keyParam, iv));
200
201            if(forEncryption){
202                int index = inOff;
203                int outOff = 0;
204
205                out = new byte[inLen + macSize];
206                calculateMac(in, inOff, inLen, macBlock);
207                ctrCipher.processBlock(macBlock, 0, macBlock, 0);
208
209                while(index < inLen - blockSize){
210                    ctrCipher.processBlock(in, index, out, outOff)
                           ↪ ;
211                    outOff += blockSize;
212                    index += blockSize;
213                }
214
215                byte[] block = new byte[blockSize];
216                System.arraycopy(in, index, block, 0, inLen -
                       ↪ index);
217                ctrCipher.processBlock(block, 0, block, 0);
218                System.arraycopy(block, 0, out, outOff, inLen -
                       ↪ index);
219                outOff += inLen - index;
220                System.arraycopy(macBlock, 0, out, outOff, out.
                       ↪ length - outOff);
```

```
221             }
222
223             else{
224                 int index = inOff;
225                 int outOff = 0;
226                 out = new byte[inLen - macSize];
227                 System.arraycopy(in, inOff + inLen - macSize,
                        ↪ macBlock, 0, macSize);
228
229                 ctrCipher.processBlock(macBlock, 0, macBlock, 0);
230
231                 for(int i = macSize; i != macBlock.length; i++){
232                     macBlock[i] = 0;
233                 }
234
235                 while(outOff < out.length - blockSize){
236                     ctrCipher.processBlock(in, index, out, outOff)
                            ↪ ;
237                     outOff += blockSize;
238                     index += blockSize;
239                 }
240
241                 byte[] block = new byte[blockSize];
242                 System.arraycopy(in, index, block, 0, out.length -
                        ↪  outOff);
243                 ctrCipher.processBlock(block, 0, block, 0);
244                 System.arraycopy(block, 0, out, outOff, out.length
                        ↪  - outOff);
245
246                 byte[] calculatedMacBlock = new byte[blockSize];
247                 calculateMac(out, 0, out.length,
                        ↪ calculatedMacBlock);
248
249                 if(!Arrays.constantTimeAreEqual(macBlock,
                        ↪ calculatedMacBlock))
250                     throw new InvalidCipherTextException("mac
                            ↪ check in CCM failed!");
251             }
252             return out;
253         }
254
255 }
```

Listing E.3: CCM code implementation